62304644 片山さくら

2024 年 12 月 26 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　プログラムの動作

プログラム全体の機能は、順列の生成、相異なる二分探索木の生成、二分探索木の高さの平均・分散の算出、二分木の描画である。このプログラムはユーザーと標準入力やウィンドウを通して対話しながら動作する。

1.4　プログラム中で使用する構造体の説明

Node 構造体に自身のキー、左右の子ノードへのポインタを持たせることで二分探索木を実装した。binary_tree.c に定義される関数を用いて Node オブジェクトを生成、二分探索木として構成する事ができる。また、Node オブジェクトのグラフィクス的な側面として NodeGraphic 構造体を定義した。NodeGraphic オブジェクトは文字の色やエッジの色を保持し、tree_graphic.c に定義される関数を用いて、構成済みの二分探索木を描画することができる。Node オブジェクトは NodeGraphic オブジェクトに依存せず生成や木の構成が可能である一方、NodeGraphic オブジェクトは Node オブジェクトをもとに生成される。

1.5　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     for (int i = 0; i < fact;i++) {
170             Tree current = create_tree(plist[i][0]);
171             Tree* current_p = &current;
172             int current_id = 0;
173             List* keyrecord = create_list(n);
174
175
176             for (int j = 1; j < n; j++) insert_node(current_p, create_tree(plist[i][j]));
177
178             get_key_record(current, keyrecord);
179             current_id = permutation_num(keyrecord->list, keyrecord->len);
180
181             delete_list(keyrecord);
182
183             if (is_exist[current_id]) {
184                     free(current);
185                     continue;
186             }
187             else {
188                     is_exist[current_id] = 1;
189                     result[different_tree_num] = current;
190                     different_tree_num++;
191             }
192     }
193
194     for (int i = 0; i < fact; i++) free(plist[i]);
```

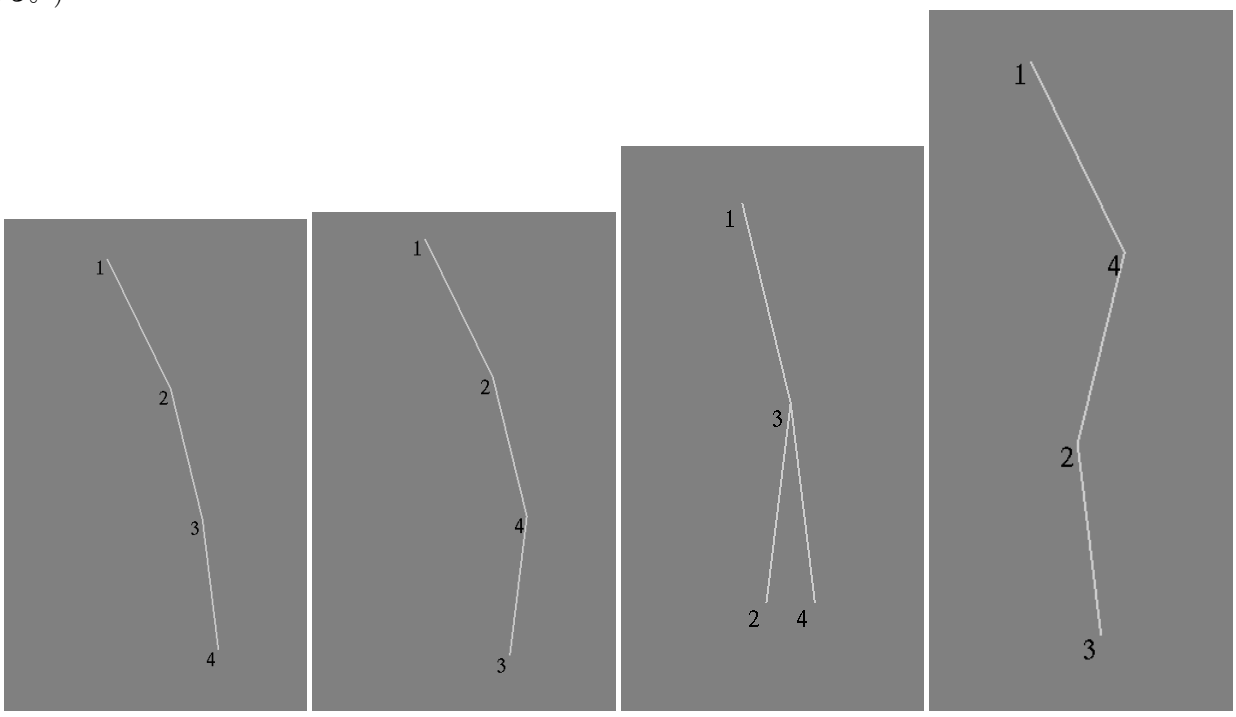
```
195         free(plist);
196         free(is_exist);
197
198         return result;
199 }
```

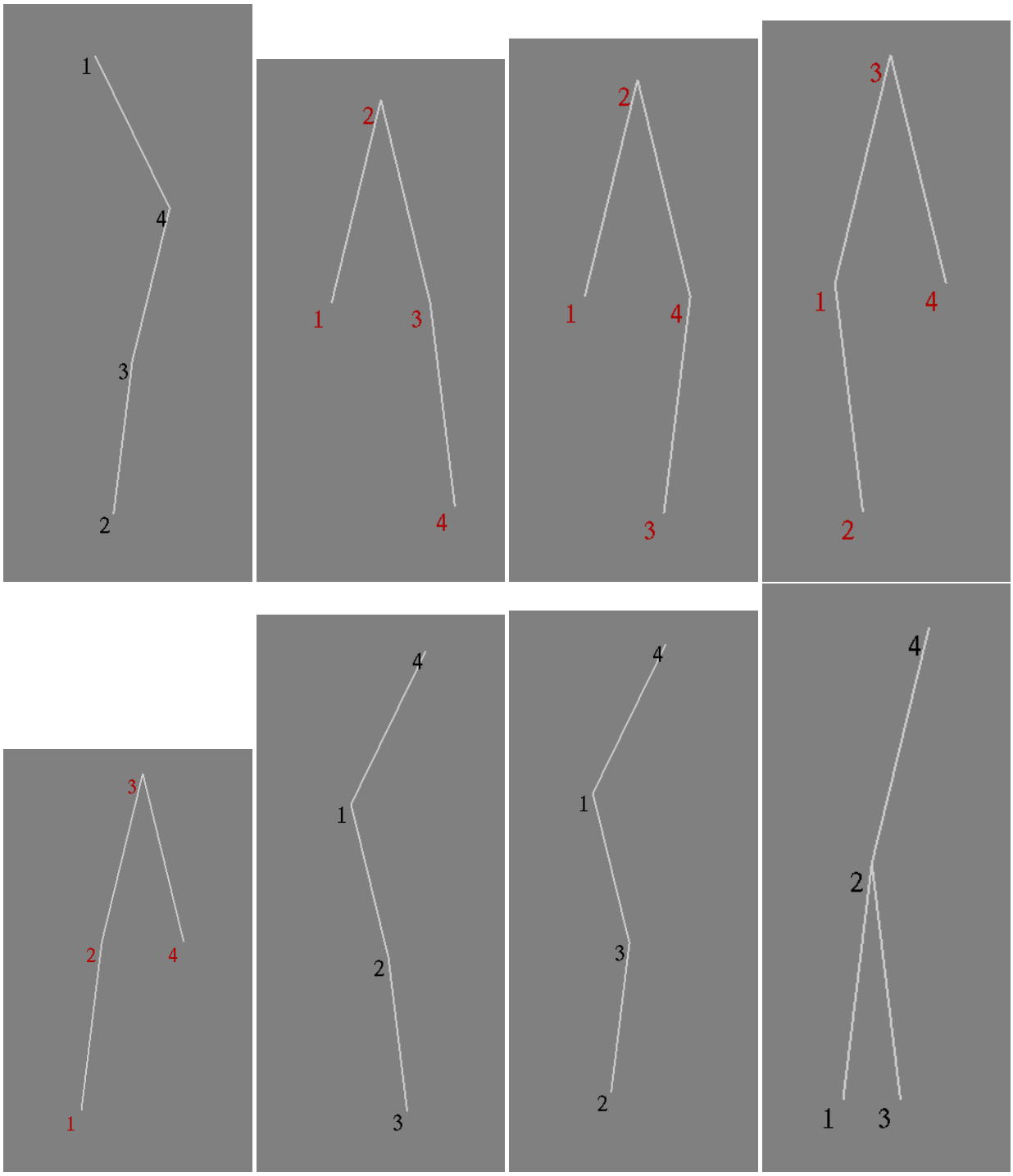
binary_tree.c において重要な関数は以下のとおりである。

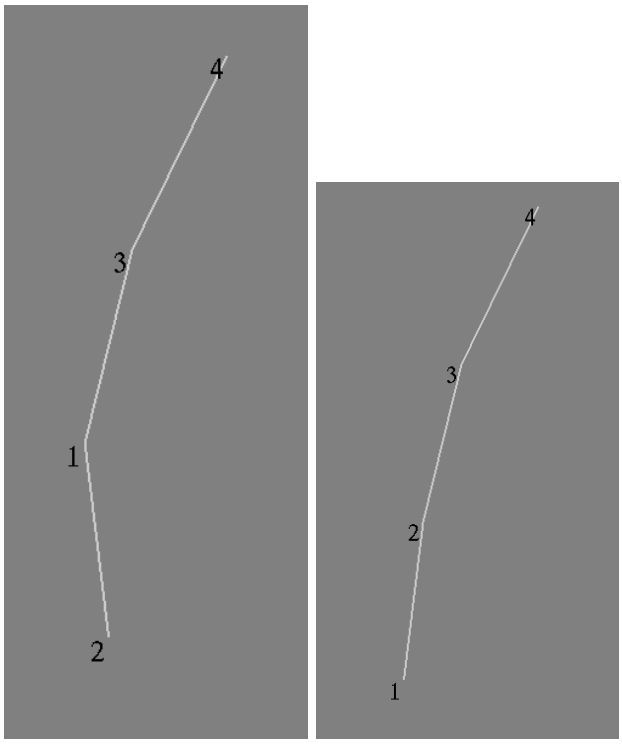
- get_key_record : あるノードを根とした部分木を前順に探索しキーの値を配列に格納する関数
- get_different_trees : 整数 n までの順列を順に二分探索木に入力し、生成したすべての相異なる二分探索木の根へのポインタをリストに格納する関数

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を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、完全二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。)



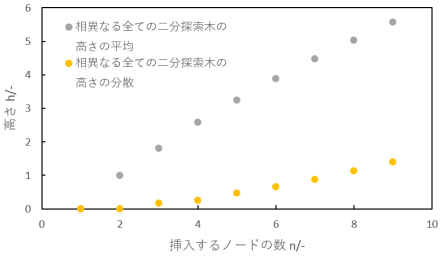

1
4
3
2
2
1
3
4
2
1
4
3
3
1
4
2



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

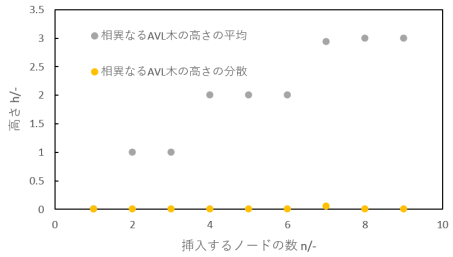
- is_complete_binary_tree： あるノードを根とした部分木が完全二分木であるかを判定する関数

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。

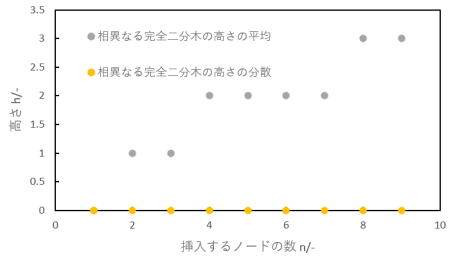
ACL 木の場合

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

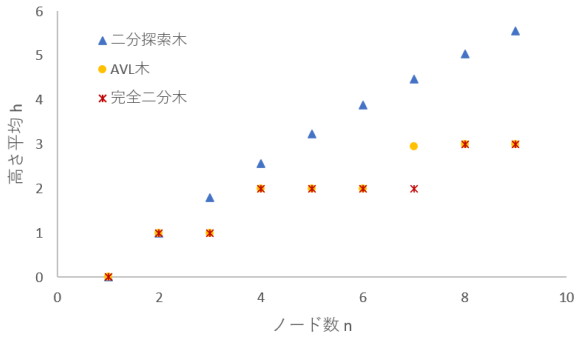


完全二分木の場合

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

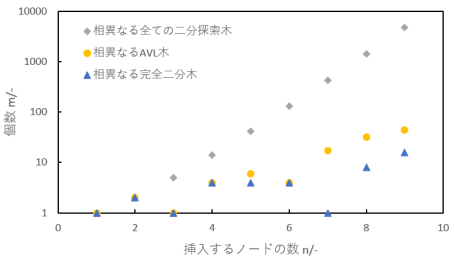


②と③の結果をまとめそれぞれの高さの平均について比較したグラフをいかに示す。



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



5 ソースコード

5.1 ソースファイル

Listing 3 binary$_t$ree.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     for (int i = 0; i < fact;i++) {
170             Tree current = create_tree(plist[i][0]);
171             Tree* current_p = &current;
172             int current_id = 0;
173             List* keyrecord = create_list(n);
174
175
176             for (int j = 1; j < n; j++) insert_node(current_p, create_tree(plist[i][j]));
177
178             get_key_record(current, keyrecord);
179             current_id = permutation_num(keyrecord->list, keyrecord->len);
180
181             delete_list(keyrecord);
182
183             if (is_exist[current_id]) {
184                     free(current);
185                     continue;
186             }
187             else {
188                     is_exist[current_id] = 1;
189                     result[different_tree_num] = current;
190                     different_tree_num++;
191             }
192     }
193
```

```
194         for (int i = 0; i < fact; i++) free(plist[i]);
195         free(plist);
196         free(is_exist);
197
198         return result;
199 }
```

Listing 4 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
46void 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
58void 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}
```

Listing 5 tree_analysis.c

```
1#include"binary_tree.h"
2#define N_MAX 9
3float all_avg;
4float all_sigma;
5float avl_avg;
6float avl_sigma;
7float cmp_avg;
8float cmp_sigma;
9int all_tree_num;
10int avl_tree_num;
```

```
11 int cmp_tree_num;
12
13 void print_hight_analysis(int tree_num, float avg, float sigma, const char* tree_name) {
14         printf(">>Analysis␣of␣hight␣of␣%s<<\n",tree_name);
15         printf("␣␣␣␣Number␣of␣trees␣␣:␣%d\n",tree_num);
16         printf("␣␣␣␣Average␣of␣hight␣:␣%f\n",avg);
17         printf("␣␣␣␣Variance␣of␣hight:␣%f\n",sigma);
18 }
19
20 void analyze_all_tree_hight(Tree* trees) {
21         float hight_sum = 0.0;
22         float hight2_sum = 0.0;
23         float sigma = 0.0;
24         int tree_num = 0;
25
26         for (; trees[tree_num] != NULL; tree_num++) {
27                 int h = calculate_hight(trees[tree_num]) - 1;
28                 hight_sum += (float)h;
29                 hight2_sum += (float)(h * h);
30         }
31
32         hight_sum /= tree_num;
33         hight2_sum /= tree_num;
34         sigma = hight2_sum - hight_sum * hight_sum;
35         print_hight_analysis(tree_num,hight_sum,sigma,"all␣distinct␣trees");
36
37         all_avg = hight_sum;
38         all_sigma = sigma;
39         all_tree_num = tree_num;
40 }
41
42 void analyze_AVL_tree_hight(Tree* trees) {
43         float hight_sum = 0.0;
44         float hight2_sum = 0.0;
45         float sigma = 0.0;
46         int tree_num = 0;
47
48         int i = 0;
49         for (i = 0; trees[i] != NULL; i++) {
50                 if (is_AVL_tree(trees[i])) {
51                         int h = calculate_hight(trees[i]) - 1;
52                         //printf("%d\n",h);
53                         hight_sum += (float)h;
54                         hight2_sum += (float)(h * h);
55                         tree_num++;
56                 }
57         }
58
59         hight_sum /= tree_num;
60         hight2_sum /= tree_num;
```

```
61          sigma = hight2_sum − hight_sum ∗ hight_sum;
62          print_hight_analysis(tree_num,hight_sum,sigma,"distinct␣AVL␣trees");
63          avl_avg = hight_sum;
64          avl_sigma = sigma;
65          avl_tree_num = tree_num;
66  }
67
68  void analyze_complete_binary_tree_hight(Tree∗ trees) {
69          float hight_sum = 0.0;
70          float hight2_sum = 0.0;
71          float sigma = 0.0;
72          int tree_num = 0;
73
74          int i = 0;
75          for (i = 0; trees[i] != NULL; i++) {
76                  if (is_complete_binary_tree(trees[i])) {
77                          int h = calculate_hight(trees[i]) − 1;
78                          //printf("%d\n", h);
79                          hight_sum += (float)h;
80                          hight2_sum += (float)(h ∗ h);
81                          tree_num++;
82                  }
83          }
84
85          hight_sum /= tree_num;
86          hight2_sum /= tree_num;
87          sigma = hight2_sum − hight_sum ∗ hight_sum;
88          print_hight_analysis(tree_num,hight_sum,sigma,"distinct␣complete␣binary␣trees");
89          cmp_avg = hight_sum;
90          cmp_sigma = sigma;
91          cmp_tree_num = tree_num;
92  }
93
94  void analyze_all_hight_distribution(int histogram[][N_MAX], Tree∗ trees,int n) {
95          for (int i = 0; trees[i] != NULL; i++) histogram[n][calculate_hight(trees[i])−1]++;
96  }
97
98  void analyze_avl_hight_distribution(int histogram[][N_MAX], Tree∗ trees, int n) {
99          for (int i = 0; trees[i] != NULL; i++) if(is_AVL_tree(trees[i])) histogram[n][calculate_hight(trees[i])
                - 1]++;
100 }
101
102 void analyze_cmp_hight_distribution(int histogram[][N_MAX], Tree∗ trees, int n) {
103         for (int i = 0; trees[i] != NULL; i++) if(is_complete_binary_tree(trees[i])) histogram[n][
                calculate_hight(trees[i]) - 1]++;
104 }
105
106 void report_trees(const char∗ file_name) {
107         FILE∗ fp;
108         fp = fopen(file_name, "w");
```

```
109     if (fp == NULL) {
110         return;
111     }
112
113     int all_histogram[N_MAX + 1][N_MAX];
114     int avl_histogram[N_MAX + 1][N_MAX];
115     int cmp_histogram[N_MAX + 1][N_MAX];
116
117     for (int i = 0; i < N_MAX + 1; i++) for (int j = 0; j < N_MAX; j++) {
118         all_histogram[i][j] = 0;
119         avl_histogram[i][j] = 0;
120         cmp_histogram[i][j] = 0;
121     }
122
123     fprintf(fp, ",all␣avg,all␣std,avl␣avg,avl␣std,cmp␣avg,cmp␣std,,all␣num,avl␣num,cmp␣num,\n");
124
125     for (int i = 1; i <= N_MAX; i++) {
126         printf("<%d>\n",i);
127         Tree* trees = get_different_trees(i);
128
129         analyze_all_tree_hight(trees);
130         analyze_AVL_tree_hight(trees);
131         analyze_complete_binary_tree_hight(trees);
132         analyze_all_hight_distribution(all_histogram, trees, i);
133         analyze_avl_hight_distribution(avl_histogram, trees, i);
134         analyze_cmp_hight_distribution(cmp_histogram, trees, i);
135
136         fprintf(fp,"%d,%f,%f,%f,%f,%f,%f,,%d,%d,%d,\n",i,all_avg,all_sigma,avl_avg,avl_sigma,
                cmp_avg,cmp_sigma,all_tree_num,avl_tree_num,cmp_tree_num);
137
138         for (int i = 0; trees[i] != NULL; i++) delete_tree(trees[i]);
139         free(trees);
140     }
141
142     for (int i = 0; i < N_MAX; i++) {
143         for (int j = 1; j < N_MAX + 1; j++) {
144             all_histogram[0][i] += all_histogram[j][i];
145             avl_histogram[0][i] += avl_histogram[j][i];
146             cmp_histogram[0][i] += cmp_histogram[j][i];
147         }
148     }
149
150     fprintf(fp,"all␣hight,0,1,2,3,4,5,6,7,8,\n");
151
152     for (int i = 0; i < N_MAX+1; i++) {
153         if (i == 0) fprintf(fp,"total,");
154         else fprintf(fp,"%d,",i);
155
156         for (int j = 0; j < N_MAX; j++) {
157             fprintf(fp,"%d,",all_histogram[i][j]);
```

```
158                 }
159                 fprintf(fp, "\n");
160         }
161
162         fprintf(fp, "avl␣hight,0,1,2,3,4,5,6,7,8,\n");
163
164         for (int i = 0; i < N_MAX + 1; i++) {
165                 fprintf(fp, "%d,", i);
166                 for (int j = 0; j < N_MAX; j++) {
167                         fprintf(fp, "%d,", avl_histogram[i][j]);
168                 }
169                 fprintf(fp, "\n");
170         }
171
172         fprintf(fp, "cmp␣hight,0,1,2,3,4,5,6,7,8,\n");
173
174         for (int i = 0; i < N_MAX + 1; i++) {
175                 fprintf(fp, "%d,", i);
176                 for (int j = 0; j < N_MAX; j++) {
177                         fprintf(fp, "%d,", cmp_histogram[i][j]);
178                 }
179                 fprintf(fp, "\n");
180         }
181
182         fclose(fp);
183 }
```

Listing 6 tree$_g$*raphic.c*

```
1
2 #include"GL/freeglut.h"
3 #include"GLFW/glfw3.h"
4 #include"tree_graphic.h"
5 #include"position.h"
6 #include<stdio.h>
7 #include"charactor.h"
8
9
10 NodeGraphic* pNodeGraphic(void* p) {
11     return (NodeGraphic*)p;
12 }
13
14 void render_val(GLFWwindow* window, NodeGraphic node) {
15     glfwMakeContextCurrent(window);//get window active
16     char buf[10];
17     snprintf(buf, 10, "%d", node.val);
18     glColor3fv(node.char_color);
19     glRasterPos2i(node.pos.x, node.pos.y);
20     for (int i = 0; buf[i] != '\0'; i++) glutBitmapCharacter(UserFont, buf[i]);
21 }
22
```

```
23 void render_box(GLFWwindow* window, NodeGraphic node) {
24     glfwMakeContextCurrent(window);//get window active
25 
26     int val_len = 1;
27     int tmp = node.val;
28     while (tmp / 10) {
29         val_len++;
30         tmp /= 10;
31     }
32 
33     glBegin(GL_POLYGON);
34     glColor3fv(node.box_color);
35     glVertex2i(node.pos.x,node.pos.y);
36     glVertex2i(node.pos.x,node.pos.y+ UserFontSizeH);
37     glVertex2i(node.pos.x+UserFontSizeW*val_len,node.pos.y+UserFontSizeH);
38     glVertex2i(node.pos.x+UserFontSizeW*val_len,node.pos.y);
39     glEnd();
40 }
41 
42 void render_edge(GLFWwindow* window,NodeGraphic a,NodeGraphic b) {
43     glfwMakeContextCurrent(window);//get window active
44 
45     glLineWidth(2);
46     glBegin(GL_LINES);
47     glColor3fv(a.edge_color);
48     glVertex2i(a.pos.x+UserFontSizeW/2,a.pos.y+UserFontSizeH/2);
49     glColor3fv(b.edge_color);
50     glVertex2i(b.pos.x+UserFontSizeW/2, b.pos.y+UserFontSizeH/2);
51     glEnd();
52 }
53 
54 
55 void render_node(GLFWwindow* window,NodeGraphic node){
56     glfwMakeContextCurrent(window);//get window active
57     //render_box(window,node);
58     render_val(window,node);
59     glFlush();
60 }
61 
62 void render_tree(GLFWwindow* window,Tree current,Tree parent) {
63     if (current == NULL) return;
64     render_tree(window,current->left,current);
65     render_tree(window,current->right,current);
66 
67     render_edge(window, *pNodeGraphic(current->pointer), *pNodeGraphic(parent->pointer));
68     render_node(window,*pNodeGraphic(current->pointer));
69 
70 }
71 
72 void update_tree_position(Tree root,int width,int hight) {
```

```
73      //printf("<%d,%d>\n",pNodeGraphic(root->pointer)->pos.x, pNodeGraphic(root->pointer)->pos.y);
74      if(root->left != NULL) {
75          pNodeGraphic(root->left->pointer)->pos.x = -width + pNodeGraphic(root->pointer)->pos.x;
76          pNodeGraphic(root->left->pointer)->pos.y = -hight + pNodeGraphic(root->pointer)->pos.y;
77          //printf("<%d,%d>\n", pNodeGraphic(root->left->pointer)->pos.x, pNodeGraphic(root->left->
                pointer)->pos.y);
78          update_tree_position(root->left, (int)(width / 2), hight);
79      }
80      if(root->right != NULL) {
81          pNodeGraphic(root->right->pointer)->pos.x = width + pNodeGraphic(root->pointer)->pos.x;
82          pNodeGraphic(root->right->pointer)->pos.y = -hight + pNodeGraphic(root->pointer)->pos.y;
83          //printf("<%d,%d>\n", pNodeGraphic(root->right->pointer)->pos.x, pNodeGraphic(root->right
                ->pointer)->pos.y);
84          update_tree_position(root->right, (int)(width / 2), hight);
85      }
86  }
87
88  int calculate_tree_min_width(Tree root, int min_dist) {
89      int tree_hight = calculate_hight(root) - 1;
90      int tree_width = 0;
91      for (int i = 0; i < tree_hight; i++) tree_width = tree_width * 2 + min_dist;
92      return tree_width + min_dist;
93  }
94
95  XYi set_tree_position(Tree root,int hight,int min_dist) {
96      XYi tree_size;
97      int tree_min_width = calculate_tree_min_width(root, min_dist);
98      int tree_hight = calculate_hight(root) - 1;
99      pNodeGraphic(root->pointer)->pos.x = 0;
100     pNodeGraphic(root->pointer)->pos.y = 0;
101     update_tree_position(root, tree_min_width / 4, hight);
102     tree_size.x = tree_min_width;
103     tree_size.y = hight * tree_hight;
104     return tree_size;
105 }
106
107 void set_tree_color(Tree root, float* char_color, float* edge_color, float* box_color) {
108     if (root == NULL) return;
109     set_tree_color(root->left, char_color, edge_color, box_color);
110     set_tree_color(root->right, char_color, edge_color, box_color);
111
112     for (int i = 0; i < 3; i++) (pNodeGraphic(root->pointer)->char_color)[i] = char_color[i];
113     for (int i = 0; i < 3; i++) (pNodeGraphic(root->pointer)->edge_color)[i] = edge_color[i];
114     for (int i = 0; i < 3; i++) (pNodeGraphic(root->pointer)->box_color)[i] = box_color[i];
115 }
116
117
118 void initialize_graphical_tree(Tree root) {
119     if (root == NULL) return;
120     initialize_graphical_tree(root->left);
```

```
121     initialize_graphical_tree(root->right);
122     NodeGraphic* new_ = (NodeGraphic*)malloc(sizeof(NodeGraphic));
123     root->pointer = (void*)new_;
124 }
125
126 void delete_graphical_tree(Tree root) {
127     if (root == NULL) return;
128     delete_graphical_tree(root->left);
129     delete_graphical_tree(root->right);
130     if (root->pointer == NULL) return;
131     free(root->pointer);
132 }
133
134 void copy_key_on_graphic_val(Tree root) {
135     if (root == NULL) return;
136     copy_key_on_graphic_val(root->left);
137     copy_key_on_graphic_val(root->right);
138     if (root->pointer == NULL) return;
139     pNodeGraphic(root->pointer)->val = root->key;
140 }
```

Listing 7 simulation_graphic.c

```
1 #include"GL/freeglut.h"
2 #include"GLFW/glfw3.h"
3 #include <stdio.h>
4 #include<stdlib.h>
5 #include"list.h"
6 #include"binary_tree.h"
7 #include"utility.h"
8 #include"tree_analysis.h"
9 #include"tree_graphic.h"
10 #include"simulation_graphic.h"
11 #include"tree_graphic.h"
12
13 GLFWwindow* window1;
14 GLFWwindow* window2;
15
16 int window_hight = WINDOW_HIGHT;
17 int window_width = WINDOW_WIDTH;
18
19 int window2_origin_x = 0;
20 int window2_origin_y = 0;
21 int window2_box_x = WINDOW_HIGHT;
22 int window2_box_y = WINDOW_WIDTH;
23
24 Tree* tree_list;
25 Tree current_tree_on_display;
26
27 int current_tree = -1;
28
```

```
29 void update_current_tree_on_display(GLFWwindow* window, Tree root) {
30         if (root == NULL) return;
31         current_tree_on_display = root;
32         float char_color[3] = { 0.0,0.0,0.0 };
33         float edge_color[3] = { 0.7,0.7,0.7 };
34         float box_color[3] = { 1.0,1.0,1.0 };
35 
36         if (is_AVL_tree(root)) {
37                 char_color[0] = 0.0;
38                 char_color[1] = 0.0;
39                 char_color[2] = 0.7;
40         }
41         if (is_complete_binary_tree(root)) {
42                 char_color[0] = 0.7;
43                 char_color[1] = 0.0;
44                 char_color[2] = 0.0;
45         }
46 
47         initialize_graphical_tree(current_tree_on_display);
48         copy_key_on_graphic_val(current_tree_on_display);
49         XYi tree_size = set_tree_position(current_tree_on_display, 100, 16);
50         set_tree_color(current_tree_on_display, char_color, edge_color, box_color);
51 
52         window2_origin_x = -TREE_POSITION_OFFSET_X - tree_size.x / 2;
53         window2_origin_y = -TREE_POSITION_OFFSET_Y -tree_size.y;
54 
55         window2_box_x = tree_size.x / 2 + TREE_POSITION_OFFSET_X;
56         window2_box_y = tree_size.y + TREE_POSITION_OFFSET_Y;
57 }
58 
59 Tree* generate_trees() {
60         int num;
61         printf("Input␣a␣number>␣");
62         do {} while (scanf("%d", &num) != 1);
63         Tree* trees = get_different_trees(num);
64 
65         analyze_all_tree_hight(trees);
66         analyze_AVL_tree_hight(trees);
67         analyze_complete_binary_tree_hight(trees);
68 
69         printf("Click␣left␣or␣right␣button\n");
70         printf("You␣can␣browse␣all␣trees␣when␣you␣push␣right␣button\n");
71         printf("You␣can␣browse␣AVL␣trees␣when␣you␣push␣left␣button\n");
72         printf("␣␣␣␣␣␣␣␣AVL␣trees␣have␣blue␣charactors\n");
73         printf("␣␣␣␣␣␣␣␣Complete␣binary␣trees␣have␣red␣charactors\n");
74 
75         return trees;
76 }
77 
78 
```

```
79 void render1(int w, int h) {
80         glViewport(0, 0, w, h);
81         glMatrixMode(GL_PROJECTION);
82         glLoadIdentity();
83         gluOrtho2D(0.0, (double)w, 0.0, (double)h);
84         window_hight = h;
85         window_width = w;
86 
87         glClearColor(0.0, 0.21, 0.38, 1.0);
88         glClear(GL_COLOR_BUFFER_BIT);
89 
90         glFlush();
91 }
92 
93 void render2(int w, int h) {
94         glViewport(0, 0, w, h);
95         glMatrixMode(GL_PROJECTION);
96         glLoadIdentity();
97         gluOrtho2D(window2_origin_x, (double)(-window2_origin_x), window2_origin_y, (double
               )(100));
98 
99 
100        glClearColor(0.5, 0.5, 0.5, 0.5);
101        glClear(GL_COLOR_BUFFER_BIT);
102 
103        render_tree(window2, current_tree_on_display, current_tree_on_display);
104 
105        glFlush();
106 }
107 
108 void mouse_callback(GLFWwindow* pwin, int button, int action, int mods) {
109        if (action != GLFW_PRESS) {
110                return;
111        }
112 
113        if (current_tree == -1) {
114                tree_list = generate_trees();
115                current_tree = 0;
116                return;
117        }
118 
119 
120        if (pwin == window2) {
121                if (tree_list[current_tree] == NULL) {
122                        for (int i = 0; tree_list[i] != NULL; i++) delete_tree(tree_list[i]);
123                        free(tree_list);
124                        tree_list = generate_trees();
125                        current_tree = 0;
126                }
127                else {
```

```
128                         glClearColor(0.3, 0.21, 0.38, 1.0);
129                         glClear(GL_COLOR_BUFFER_BIT);
130                         delete_graphical_tree(current_tree_on_display);
131                 }
132                 printf("[%d]\n",current_tree);
133                 if (button == GLFW_MOUSE_BUTTON_LEFT) {
134                         while (tree_list[current_tree] != NULL && is_AVL_tree(tree_list[current_tree])
                                == 0) current_tree++;
135                 }
136                 update_current_tree_on_display(window2, tree_list[current_tree]);
137                 current_tree++;
138         }
139 }
140
141 void glfwMainLoop() {
142         while (glfwWindowShouldClose(window2) == GL_FALSE) {
143                 int width, height;
144
145                 glfwGetFramebufferSize(window2, &width, &height);
146                 glfwMakeContextCurrent(window2);
147                 render2(width, height);
148                 glfwSwapBuffers(window2);
149
150                 glfwWaitEvents();
151         }
152 }
153
154 void init_OpenGL(int argc, char** argv) {
155         glutInit(&argc, argv);
156         if (glfwInit() == GL_FALSE) return 0;
157         window2 = glfwCreateWindow(WINDOW_HIGHT, WINDOW_WIDTH, "display_window", NULL,
                NULL);
158
159         glfwSetMouseButtonCallback(window2, mouse_callback);
160
161 }
```

Listing 8 main.c

```
1 #include"simulation_graphic.h"
2 #include"binary_tree.h"
3 #include"tree_analysis.h"
4 const char* file_name = "result.csv";
5
6 int main(int argc, char** argv) {
7         char mode;
8         printf("Select␣mode␣[a/b]\na␣-␣analize␣binary␣trees␣and␣make␣csv␣file.\nb␣-␣start␣
              graphical␣mode\n");
9         if (scanf("%c", &mode) == EOF) {
10                printf("invalid␣input\n");
11                return 0;
```

```
12      }
13          if (mode == 'a') {
14                  report_trees(file_name);
15          }
16          else if(mode == 'b'){
17                  init_OpenGL(argc, argv);
18                  printf("Click␣display_window\n");
19                  glfwMainLoop();
20                  glfwTerminate();
21          }
22  }
```

Listing 9 list.c

```
1 #include"list.h"
2 #include"utility.h"
3
4 //--------------list----------------
5 List* create_list(int lim){
6           List* new_ = (List*)malloc(sizeof(List));
7           if (new_ == NULL) return NULL;
8           new_->list = (int*)calloc(sizeof(int),lim);
9           if (new_->list == NULL) return NULL;
10          new_->size = lim;
11          new_->len = 0;
12          return new_;
13  }
14
15 void delete_list(List* list){
16          free(list->list);
17          free(list);
18  }
19
20 int push_back_list(List* list,int num){
21          if(list->size == list->len) return 0;
22          list->list[list->len] = num;
23          list->len += 1;
24          return 1;
25  }
26
27 void print_list(List list){
28          for(int i = 0;i<min_(list.size,list.len);i++)printf("%d␣",list.list[i]);
29          putchar('\n');
30  }
```

Listing 10 position.c

```
1 #include"position.h"
2
3 XYi add_xyi(XYi a,XYi b){
4       XYi res;
```

```
5     res.x = a.x + b.x;
6     res.y = a.y + b.y;
7     return res;
8 }
```

5.2 ヘッダファイル

Listing 11 binary$_t ree.h$

```
1 #ifndef BINARY_TREE_H_
2 #define BINARY_TREE_H_
3 #include<stdio.h>
4 #include<stdlib.h>
5 #include"list.h"
6
7 struct Node{
8     int key;
9     struct Node* left;
10    struct Node* right;
11    void* pointer;
12 };
13
14 typedef struct Node* Tree;
15
16 typedef struct tree_list{
17          Tree* list;
18          int size;
19          int len;
20 }TreeList;
21
22 struct Node* create_tree(int key);
23 void delete_tree(Tree node);
24 int count_children(Tree root);
25 int calculate_hight(Tree root);
26 int is_AVL_tree(Tree root);
27 int is_complete_binary_tree(Tree root);
28 void insert_node(Tree* root,Tree node);
29 Tree extract_max(Tree root);
30 void delete_node_from_tree(Tree* root,int key);
31 void search_pre(Tree root);
32 void search_pre_with_depth(Tree root,int depth);
33 void search_middle(Tree root);
34 void search_post(Tree root);
35 void print_tree(Tree root);
36
37 void get_key_record(Tree root,List* list);
38 Tree* get_different_trees(int n);
39
40 #endif
```

Listing 12　utility.h

```
#ifndef UTILITY_H_
#define UTILITY_H_

int max_(int a,int b);
int min_(int a,int b);
int factorial(int n);
int permutation_num(int* num,int n);
void clear(int *array,int n);
int** get_permutation_list(int n);
void set_permutation_list(int** list, int n);

#endif
```

Listing 13　tree$_a$*nalysis.h*

```
#ifndef TREE_ANALYSIS
#define TREE_ANALYSIS
#include"binary_tree.h"

void analyze_all_tree_hight(Tree* trees);
void analyze_AVL_tree_hight(Tree* trees);
void analyze_complete_binary_tree_hight(Tree* trees);
void report_trees(const char* file_name);
#endif
```

Listing 14　tree$_g$*raphic.h*

```
#ifndef TREE_GRAPHIC_H_
#define TREE_GRAPHIC_H_

#include"GL/freeglut.h"
#include"GLFW/glfw3.h"
#include"binary_tree.h"
#include"position.h"
#include"charactor.h"

#define TREE_POSITION_OFFSET_X 100
#define TREE_POSITION_OFFSET_Y 100

typedef struct node_graphic{
    XYi pos;
    float box_color[3];
    float char_color[3];
    float edge_color[3];
    int val;
}NodeGraphic;

typedef struct tree_graphic{
    Tree binary_tree;
    int edge_thin;
```

```
24 }TreeGraphic;
25
26 NodeGraphic* pNodeGraphic(void* p);
27
28 void render_node(GLFWwindow* window,NodeGraphic node);
29
30 void update_tree_position(Tree root, int width, int hight);
31
32 XYi set_tree_position(Tree root, int hight, int min_dist);
33
34 void set_tree_color(Tree root, float* char_color, float* edge_color, float* box_color);
35
36 void render_tree(GLFWwindow* window, Tree current, Tree parent);
37
38 void initialize_graphical_tree(Tree root);
39
40 void delete_graphical_tree(Tree root);
41
42 void copy_key_on_graphic_val(Tree root);
43
44 void print_pos(Tree root);
45
46 #endif
```

Listing 15 simulation$_g$*raphic.h*

```
1 #ifndef SIMULATION_GRAPHIC_H_
2 #define SIMULATION_GRAPHIC_H_
3 #include"binary_tree.h"
4 #include"tree_graphic.h"
5
6 #define WINDOW_HIGHT 800
7 #define WINDOW_WIDTH 800
8
9 extern Tree current_tree_on_display;
10 extern Tree* tree_list;
11
12 void glfwMainLoop();
13
14 void init_OpenGL(int argc, char** argv);
15
16 #endif
```

Listing 16 list.h

```
1 #ifndef LIST_H_
2 #define LIST_H_
3 #include<stdio.h>
4 #include<stdlib.h>
5
6 typedef struct list{
```

```
 7         int* list;
 8         int size;
 9         int len;
10 }List;
11
12 List* create_list(int lim);
13 void delete_list(List* list);
14 int push_back_list(List* list,int num);
15 void print_list(List list);
16
17 #endif
```

Listing 17　position.h

```
1 #ifndef POSITION_H_
2 #define POSITION_H_
3
4 typedef struct xyi{
5     int x;
6     int y;
7 }XYi;
8
9 XYi add_xyi(XYi a,XYi b);
10
11 #endif
```

最新のコードは以下のリンクから取得してください。

`https://github.com/zenon-paul/opengl_binary_tree`