データ構造とアルゴリズム実験レポート
課題:2 分木、2 分探索木

202110796 4 クラス 高橋大粋

締切日:2024 年 11 月 18 日
2024 年 11 月 18 日

1　基本課題

この課題では、教科書リスト 2.11 から 2.12(p.40 から 41) の二分木の C プログラム binarytree.c と、教科書リスト 4.3 から 4.7(p.69 から 77) の二分探索木の C プログラム binarysearchtree.c のリストおよび実行結果を示した。

1.1　2 分木の実装

1.1.1　実装の方針

まず、二分木の機能として label を節点のラベル、左部分木の根を left、右部分木の根を right とする二分木を作成し、作成した節点のポインタを返す create_node()、節点 n を根とする二分木を探索し、行きがけ順で節点のラベルを標準出力に表示する preorder()、節点 n を根とする二分木を探索し、通りがけ順で節点のラベルを標準出力に表示する inorder()、節点 n を根とする二分木を探索し、帰りがけ順で節点のラベルを標準出力に表示する postorder()、節点 n を根とする二分木を幅優先探索し、節点のラベルを標準出力に表示する breadth_first_search()、節点 n を根とする二分木の構造が完全にわかる形式で標準出力に表示する display()、節点 n を根とする二分木の高さを返す height()、節点 n を根とする二分木を削除する delete_tree() を binarytree.c に実装した。また、main 関数は別ファイル main_binarytree.c に実装した。

1.1.2　実装コードおよびコードの説明

プログラム 1 に、binarytree.c の主要部を示す。

Node *create_node(char *label,Node *left,Node *right) 関数は、節点とラベルのメモリを確保し、label、left、right それぞれに節点のラベル、左部分木の根、右部分木の根を割り当て初期化し、節点のポインタを返す。void preorder(Node *n) 関数は、方式識別のための文字列 PRE: を表示するために static int is_first_call を用いて関数が最初に呼び出されたときだけ PRE: を表示する。また、この関数では再帰を用いて行きがけ順での木の走査を実装しているが、再帰の終了後に改行するために root に根を記憶しておき、初回呼び出しでなく引数が根のときに再帰が終了したとみなし、改行する。void inorder(Node *n) 関数は、preorder と同様に初回呼び出しで IN: を表示し、通りがけ順で木の走査をし、再帰終了後に改行する。void postorder(Node *n) 関数は、preorder と同様に初回呼び出しで POST: を表示し、帰りがけ順で木の走査をし、再帰終了後

に改行する。void display(Node *n) 関数は、preorder と同様に初回呼び出しで TREE: を表示し、ノードが NULL なら null を、ノードにラベルがあるなら label(を表示する。この関数でも再帰を用いており、左の子に移動した後に, を、右の子に移動した後に) を表示することにより、X(L,R) という形式で二分木の構造が完全にわかるように表示できる。ここで X はラベル L は左部分木、R は右部分木を表す。また、再帰終了後に改行する。void breadth_first_search(Node *n) 関数は、課題 2 で実装したキューを Node 型を扱えるように改良して利用している。サイズが 100 のキューを作成し、ルートノードをキューに追加し、BFS: を表示する。キューがからになるまで次を繰り返す。キューの先頭からノードを取り出す、そのノードのラベルを出力、左の子ノードが存在すればキューに追加、右の子ノードが存在すればキューに追加。探索が終了したら改行し、キューを開放する。int height(Node *n) 関数は、節点 n が NULL なら高さ 0 を返し、根のみの場合は高さ 1 を返す。左部分木と右部分木それぞれで再帰を用いて高さを計算し、大きい方の値 +1 を高さとして返す。void delete_tree(Node *n) 関数は、再帰を用いて節点 n とその子が確保したメモリをすべて開放する。また、ラベルが確保したメモリも開放する。

なお、これらの関数では節点が NULL の場合は基本的に何もせず終了する。

```
1  Node *create_node(char *label, Node *left, Node *right){
2      Node *node = (Node *)malloc(sizeof(Node));
3      if (node == NULL) {
4          fprintf(stderr, "メモリの割り当てに失敗しました\n");
5          exit(EXIT_FAILURE);
6      }
7       node->label = (char *)malloc(strlen(label) + 1);
8      if (node->label == NULL) {
9          fprintf(stderr, "文字列用のメモリ割り当てに失敗しました\n");
10         free(node);
11         exit(EXIT_FAILURE);
12     }
13     strcpy(node->label, label);
14     node->left = left;
15     node->right = right;
16     return node;
17 }
18
19 void preorder(Node *n){
20     static int is_first_call = 1;
21     static Node *root = NULL;
22
23     if (is_first_call) {
24         printf("PRE: ");
25         is_first_call = 0;
26         root = n;
```

```
27      }
28      if (n == NULL) return;
29      printf("%s ", n->label);
30      preorder(n->left);
31      preorder(n->right);
32      if(is_first_call == 0 && n == root) {
33          printf("\n");
34      }
35  }
36
37  void inorder(Node *n){
38      static int is_first_call = 1;
39      static Node *root = NULL;
40
41      if(is_first_call){
42          printf("IN: ");
43          is_first_call = 0;
44          root = n;
45      }
46      if (n == NULL) return;
47      inorder(n->left);
48      printf("%s ", n->label);
49      inorder(n->right);
50      if (is_first_call == 0 && n == root) {
51          printf("\n");
52      }
53  }
54
55  void postorder(Node *n){
56      static int is_first_call = 1;
57      static Node *root = NULL;
58
59      if (is_first_call) {
60          printf("POST: ");
61          root = n;
62          is_first_call = 0;
63      }
64      if (n == NULL) return;
65      postorder(n->left);
66      postorder(n->right);
```

```
67      printf("%s ", n->label);
68      if (is_first_call == 0 && n == root) {
69          printf("\n");
70      }
71  }
72
73  void display(Node *n){
74      static int is_first_call = 1;
75      static Node *root = NULL;
76
77      if (is_first_call) {
78          printf("TREE: ");
79          root = n;
80          is_first_call = 0;
81      }
82
83      if (n == NULL) {
84          printf("null");
85          return;
86      }
87
88      printf("%s(", n->label);
89      display(n->left);
90      printf(",");
91      display(n->right);
92      printf(")");
93
94      if (is_first_call == 0 && n == root) {
95          printf("\n");
96          is_first_call = 1;
97      }
98  }
99
100 void breadth_first_search(Node *n){
101     if (n == NULL) return;
102
103     Queue *q = create_queue(100);
104     enqueue(q, n);
105
106     printf("BFS: ");
```

```
107     while (q->front != q->rear) {
108         Node *current = dequeue(q);
109         printf("%s ", current->label);
110 
111         if (current->left != NULL) enqueue(q, current->left);
112         if (current->right != NULL) enqueue(q, current->right);
113     }
114     printf("\n");
115     delete_queue(q);
116 }
117 
118 int height(Node *n){
119     if (n == NULL) return 0;
120     int left_height = height(n->left);
121     int right_height = height(n->right);
122     return 1 + (left_height > right_height ? left_height : right_height);
123 }
124 
125 void delete_tree(Node *n){
126     if (n == NULL) return;
127     delete_tree(n->left);
128     delete_tree(n->right);
129     free(n->label);
130     free(n);
131 }
```

プログラム 1　binarytree.c の主要部

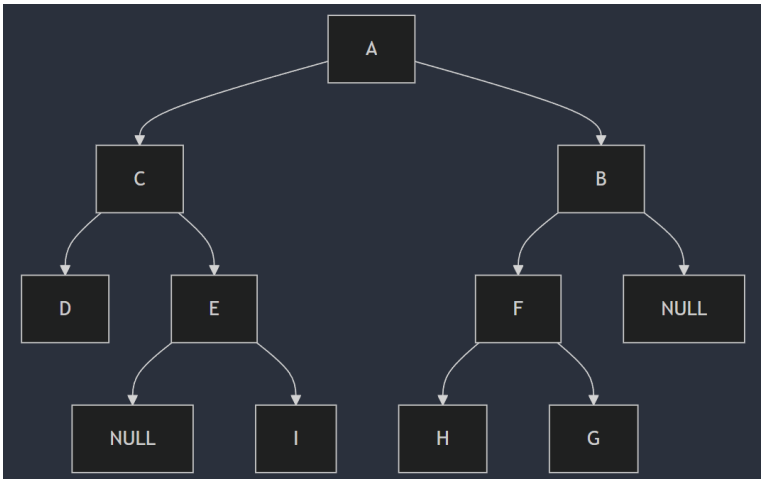
1.1.3　実行結果

まず、main_binarytree.c を以下に示す。

```
1   int main(void) {
2     // Build a binary tree
3     Node *i = create_node("I", NULL, NULL);
4     Node *h = create_node("H", NULL, NULL);
5     Node *g = create_node("G", NULL, NULL);
6     Node *d = create_node("D", NULL, NULL);
7     Node *e = create_node("E", NULL, i);
8     Node *f = create_node("F", h, g);
9     Node *c = create_node("C", d, e);
10    Node *b = create_node("B", f, NULL);
```

```
11      Node *a = create_node("A", c, b);
12
13      preorder(a);
14      inorder(a);
15      postorder(a);
16      breadth_first_search(a);
17      display(a);
18
19      printf("height: %d\n", height(a));
20
21      delete_tree(a);
22
23      return EXIT_SUCCESS;
24  }
```

今回は次の図で表す二分木に対して、実装した関数が正しく動作することを確認する。次に、binarytree.c を



以下の make コマンドを実行してコンパイルし、プログラムを実行する。

```
1   PS C:\Users\daiki\Desktop\DSA\kadai4> make
2   gcc    -c -o binarytree.o binarytree.c
3   gcc   binarytree.o main_binarytree.o   -o binarytree
4   PS C:\Users\daiki\Desktop\DSA\kadai4> ./binarytree
5   PRE: A C D E I B F H G
6   IN: D C E I A H F G B
7   POST: D I E C H G F B A
8   BFS: A C B D E F I H G
9   TREE: A(C(D(null,null),E(null,I(null,null))),B(F(H(null,null),G(null,null
        )),null))
```

```
10  height: 4
```

まず、create_node で図で表されている二分木を作成した。preorder、inorder、postorder、breadth_first_search でそれぞれラベルを表示した。図と比較してみると正しく出力されていることがわかる。display の出力も図と比較すれば正しく表示されていることがわかる。height も図を見ると木の高さが 4 なので正しく出力されている。最後に delete_tree で確保したメモリ領域を開放した。これらの関数は 1.1.2 で説明したように動作する。以上より基本課題の要件をすべて満たすことを確認した。

1.2 2 分探索木の実装

1.2.1 実装の方針

まず、2 分探索木の機能として、root を根とする二分探索木における最小値を探索し、その値を返し、二分探索木画からの場合は-1 を高さとして返す min_bst(Node *root)、整数 d が root を根とする二分探索木二存在するか検査し、存在すれば true、存在しなければ false を bool 型で返す search_bst(Node *root,int d)、root を根とする二分探索木に整数 d を追加し、二分探索木がからの場合は、整数 d の根を作成する insert_bst(Node **root,int d)、root を根とする二分探索木から整数 d を削除する delete_bst(Node **root,int d) を binarysearchtree.c に実装した。また、main 関数は別ファイル main_binarysearchtree.c に実装した。

1.2.2 実装コードおよびコードの説明

プログラム 2 に、binarysearchtree.c の主要部を示す。

int min_bst(Node *root) 関数は、葉を除く任意の節点 n に対して、その節点のデータは左部分木内の任意のデータより大きく、右部分木内の任意のデータより小さいという条件から最左端ノードのデータが最小値を取るという性質を用いて、左の子ノードが NULL 出ない限り左の子ノードに移動し続け、NULL になったときそのノードは最左端ノードなのでそのノードのデータを返す。bool search_bst(Node *root,int d) 関数は、上述した二分探索木の性質を利用して、探索対象の整数 d より p のデータが大きい場合は左部分木に、小さい場合は右部分木に移動し、値が等しい場合は探索対象が見つかったので true を返す。探索しても見つからなかった場合は false を返す。void insert_bst(Node **root,int d) 関数は、関数内で Node 型ポインタ root が保持するアドレスを書き換えるため、Node 型のポインタのポインタを第一引数にセットしている。ルートノードが空であれば、新しいノードを作成し、木のルートとして設定する。ノードの作成には 1.1 で説明した create_node を用いた。木がから出ない場合、ルートノードから始めて、次の手順で挿入位置を探索する。p ポインタで現在のノードを追跡、挿入する値 d を現在のノードの値と比較、d と等しい場合は挿入せず終了。d より大きい場合は左部分木を探索。もし左の子ノードが空なら新しいノードを挿入。d より小さい場合は右部分木を探索。もし右の子ノードが空なら新しいノードを挿入。void delete_bst(Node **root,int d) 関数は、insert_bst と同様に第一引数に Node 型ポインタのポインタを持つ。ルートノードが空の場合、何もせずに終了。そうでなければルートノードから順にノードを探索する。削除対象のノードを p、その親ノードを q に記録する。p のデータが d より大きい場合は左部分木、小さい場合は右部分木に移動する。ノードが見つからない場合は関数を終了する。p の子ノードの数に応じて次のケースを処理する。case1: 子を持たないノードの場合、親ノード q から対象ノード p を取り外し、p を開放する。もし p が根ノードの場合、ルートノードを空にする。case2:1 つの子を持つノード場合、p の子ノードを親ノード q に接続し、ノード p を開放する。もし p

が親ノードの場合、ルートノードを子ノードに更新する。case3:2 つの子を持つノードの場合、右部分木の最小値ノード s を探索し、削除対象ノード p の値を s の値で置き換える。最小値ノード s は子を持たないか、右部分木を持つため、その後 s を削除する。

```
int min_bst(Node *root){
    if(root == NULL)return -1;
    Node *p = root;
    while(p->left != NULL)p = p->left;
    return p->value;
}

bool search_bst(Node *root, int d){
    Node *p = root;
    while(p != NULL){
        if(p->value == d){
            return true;
        } else if (p->value > d){
            p = p->left;
        } else {
            p = p->right;
        }
    }
    return false;
}

void insert_bst(Node **root, int d){
    if(*root == NULL){
        *root = create_node(d, NULL, NULL);
        return;
    }
    Node *p = *root;
    while(1){
        if(p->value == d)return;
        if(p->value > d){
            if(p->left == NULL){
                p->left = create_node(d,NULL,NULL);
                return;
            } else {
                p = p->left;
            }
```

```
37          } else {
38              if(p->right == NULL) {
39                  p->right = create_node(d,NULL,NULL);
40                  return;
41              } else {
42                  p = p->right;
43              }
44          }
45      }
46  }
47
48  void delete_bst(Node **root, int d) {
49      if (*root == NULL) return;
50
51      Node *p = *root;
52      Node *q = NULL;
53
54      while (p != NULL && p->value != d) {
55          q = p;
56          if (d < p->value) {
57              p = p->left;
58          } else {
59              p = p->right;
60          }
61      }
62
63      if (p == NULL) return;
64
65  //case1
66      if (p->left == NULL && p->right == NULL) {
67          if (q == NULL) {
68              *root = NULL;
69          } else if (q->left == p) {
70              q->left = NULL;
71          } else {
72              q->right = NULL;
73          }
74          free(p);
75
76  //case2
```

```
77      } else if (p->left == NULL || p->right == NULL) {
78          Node *child = (p->left != NULL) ? p->left : p->right;
79
80          if (q == NULL) {
81              *root = child;
82          } else if (q->left == p) {
83              q->left = child;
84          } else {
85              q->right = child;
86          }
87          free(p);
88
89  //case3
90      } else {
91          Node *s = p->right;
92          Node *sp = p;
93
94          while (s->left != NULL) {
95              sp = s;
96              s = s->left;
97          }
98
99          p->value = s->value;
100
101         if (sp->left == s) {
102             sp->left = s->right;
103         } else {
104             sp->right = s->right;
105         }
106         free(s);
107     }
108 }
```

プログラム 2　binarysearchtree.c の主要部

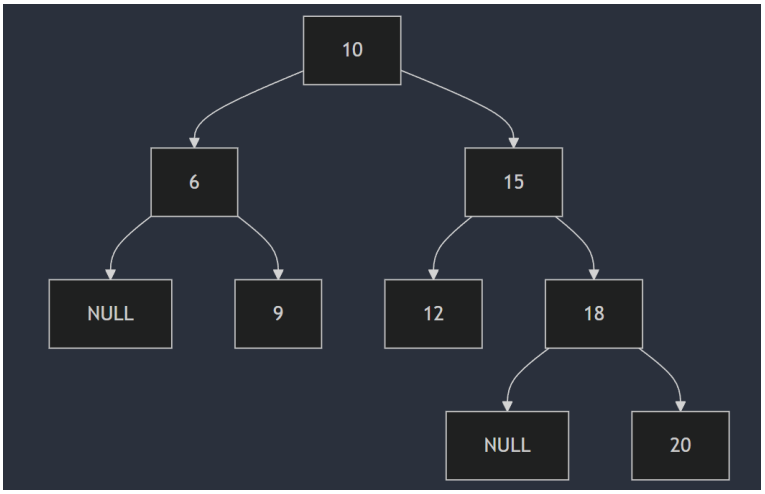
1.2.3　実行結果

まず、main_binarysearchtree.c の主要部を以下に示す。

```
1  int main(void) {
2      // Build a binary search tree
3      Node *root = NULL;
```

```
4       insert_bst(&root, 10);
5       insert_bst(&root, 15);
6       insert_bst(&root, 18);
7       insert_bst(&root, 6);
8       insert_bst(&root, 12);
9       insert_bst(&root,  20);
10      insert_bst(&root,  9);
11
12      inorder(root);
13      display(root);
14
15      printf("search_bst 10: %d\n", search_bst(root, 10));
16      printf("search_bst 12: %d\n", search_bst(root, 12));
17      printf("search_bst 15: %d\n", search_bst(root, 15));
18      printf("search_bst 7: %d\n", search_bst(root, 7));
19      printf("min_bst: %d\n", min_bst(root));
20
21      delete_bst(&root, 15);
22
23      inorder(root);
24      display(root);
25
26      delete_bst(&root, 10);
27
28      inorder(root);
29      display(root);
30
31      delete_tree(root);
32
33      root = NULL;
34      printf("search_bst 10: %d\n", search_bst(root, 10));
35
36      return EXIT_SUCCESS;
37  }
```

今回は次の図で表す二分木に対して、実装した関数が正しく動作することを確認する。次に、binary-



searchtree.c を以下の make コマンドを実行してコンパイルするために Makefile に以下の行を追加した。

```
CC:=gcc
binarytree: binarytree.o main_binarytree.o
binarysearchtree: binarysearchtree.o main_binarysearchtree.o #add
```

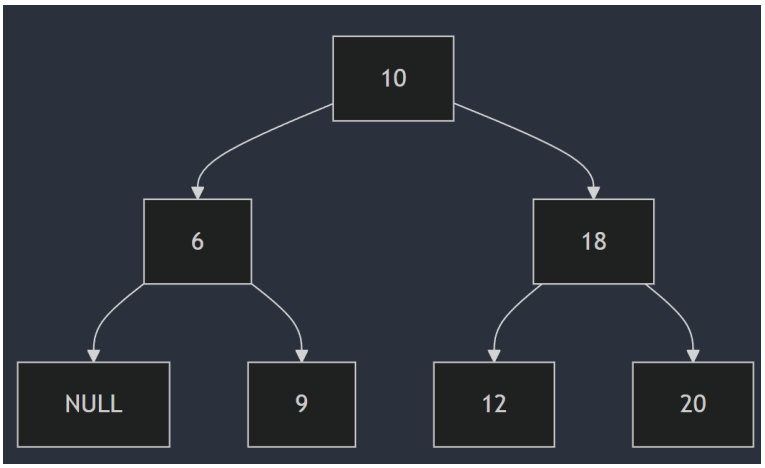
make コマンドでコンパイルし、プログラムを実行した結果は以下のようになった。

```
PS C:\Users\daiki\Desktop\DSA\kadai4> ./binarysearchtree
IN: 6 9 10 12 15 18 20
TREE: 10(6(null,9(null,null)),15(12(null,null),18(null,20(null,null))))
search_bst 10: 1
search_bst 12: 1
search_bst 15: 1
search_bst 7: 0
min_bst: 6
6 9 10 12 18 20
TREE: 10(6(null,9(null,null)),18(12(null,null),20(null,null)))
6 9 12 18 20
TREE: 12(6(null,9(null,null)),18(null,20(null,null)))
search_bst 10: 0
```

まず、insert_bst で値 10,15,18,6,12,20,9 をこの順で挿入した結果は図のような二分探索木になった。inorder で表示した結果も値の小さい順に表示されているので正しく挿入できていることが確認できる。次に 10,12,15,7 を search_bst で探索した結果を見ると、10 は根、12 は葉、15 は根以外の非終端節点、7 は二分探索木に無いデータとなっており、全て正しく探索できていることが確認できる。

delete_bst で 15 を削除した際の二分探索木を display で表示した結果が正しいかを図を用いて確認する。15 は 2 つの子を持ち、右部分木内で最も小さい値は 18 なので、18 の部分木は移動せず、15 の位置に 18 が移

動するはずである。以下にその図を示す。1.2.2 で説明したように動作するならば図のような木構造になるは



ずである。display での表示結果と見比べると木構造が一致していることがわかるので正しく削除できていることを確認した。さらに、根である 10 を削除する場合、子を 2 つ持つので、右の部分木内で最小の値である 12 が先程述べた処理同様に 10 の位置に移動する。display の表示結果を見ても 12 が根の位置に移動しているので正しく実装されていることがわかる。また、delete_tree で二分探索木を削除した後に空の木を作成し、search_bst で探索した結果、0 が表示されているので正しき探索できている。
以上の処理は 1.2.2 で説明したとおりに動作する。よってこの課題の