データ構造とアルゴリズム実験レポート

＜課題 4：2 分木、2 分探索木＞

202213025 – 3 クラス – Can Minh Nghia

締切日：2023/11/20

提出日：2023/11/20

必須課題

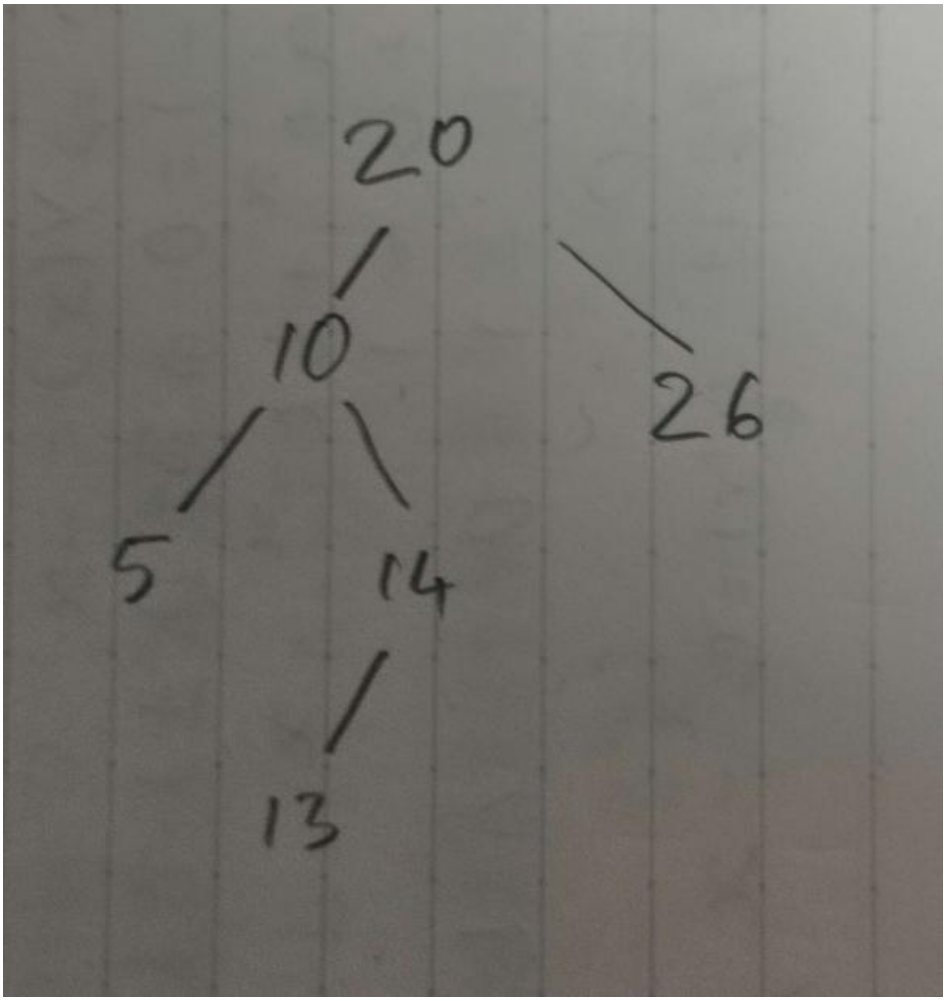
課題 4-3

4-3-1 実装の仕様

1. ラベルとして整数値をとる 2 分探索木を実現するプログラム binarysearchtree を実装せよ．binarysearchtree.c には，最低限以下の関数を定義すること．
    - `int min_bst(Node *root)`
        - root を根とする 2 分探索木における最小値を探索し，その値を返す．2 分探索木が空の場合は -1 を返す．
    - `bool search_bst(Node *root, int d)`
        - 整数 d が root を根とする 2 分探索木に存在するか検査し，存在すれば true，存在しなければ false を bool 型で返す．
    - `void insert_bst(Node **root, int d)`
        - root を根とする 2 分探索木に整数 d を追加する．2 分探索木が空の場合は，整数 d の根を作成する．
            - insert_bst 関数内で Node 型ポインタ root が保持する内容 (アドレス) を書き換えるため，insert_bst には Node 型ポインタ root のポインタ (ダブルポインタ) を第一引数にセットしている．こうしないと，insert_bst 関数内で書き換えた内容は main 関数側で反映されないので注意すること．

    - シングルポインタだと所謂「アドレスの値渡し」となるので，関数内でアドレスを変更しても，それは**関数にコピーされた値を操作しただけ**なので，関数の呼び出し側には反映されない
- `void delete_bst(Node **root, int d)`
    - root を根とする 2 分探索木から整数 d を削除する．insert_bst 関数と同様に，第一引数に Node 型ポインタ root のポインタをセットしている．

4-3-2 実装コードおよび実装コードの説明



課題の二分木

binarysearchtree.c ファイル：

ここで、新しく作った 4 つの関数があるので、1 つずつ開設する。ただし、create_tree, preorder, inorder, postorder, display, delete_tree 関数は binarytree.c ファイルのものを流用する。

- `int min_bst(Node *root)`
  - root を根とする 2 分探索木における最小値を探索し，その値を返す．2 分探索木が空の場合は -1 を返す．

```
 6 int min_bst(Node *root) {
 7       if (root == NULL) return -1;
 8       Node *min = root;
 9       while (min -> left != NULL) {
10             min = min -> left;
11       }
12
13       return min -> value;
14 }
```

Line 7: 2 分探索木が空の場合は -1 を返す。

Line 9 ~ 10：現在のノードの値より小さいノードがあればポインタ min をそこに移動する。一番小さい値を持つノードは木の左の端っこにあるから、ポインタを木の左の端っこのノードまで移動させて、そこにある値(min -> value)を返す。

- `bool search_bst(Node *root, int d)`

整数 d が root を根とする 2 分探索木に存在するか検査し，存在すれば true，存在しなければ false を bool 型で返す

```
16 bool search_bst(Node *root, int d) {
17       Node *p = root;
18
19       while (p != NULL) {
20             if (p -> value == d) {
21                   return true;
22             }
23             else if (d < p -> value) {
24                   p = p -> left;
25             } else {
26                   p = p -> right;
27             }
28       }
29       return false;
30 }
```

Line 17:まず根に指すポインタ p を作る。探したいノードの値を現在のノードの値と比較する。

Line 19 ~ 22：現在のノードの値が d と等しかったら true を返す。

Line 23~ 24：d が現在のノードの値より小さい => ポインタ p を左に移動させる。

Line 25~ 26：d が現在のノードの値より大きい => ポインタ p を右に移動させる。

Line 29：上の全部の処理が終わっても d の値を持つノードが見つからなかったら false を返す。つまり d の値を持つノードが存在しない。

- `void insert_bst(Node **root, int d)`
  - root を根とする 2 分探索木に整数 d を追加する．2 分探索木が空の場合は，整数 d の根を作成する．

insert_bst 関数内で Node 型ポインタ root が保持する内容 (アドレス) を書き換えるため，insert_bst には Node 型ポインタ root のポインタ (ダブルポインタ) を第一引数にセットしている．こうしないと，insert_bst 関数内で書き換えた内容は main 関数側で反映されないので注意すること

まず、binarytree.c の create_tree を導入する。後で insert_bts を書く。

```
40 void insert_bst(Node **root, int d) {       //double pointer works as 2 variables
41       if (*root == NULL) { //nothing is currently poiting to root
42             *root = create_tree(d, NULL, NULL);
43             return;
44       }
45
46       Node *new_node = *root;
47
48       while (1) {
49             if (d == new_node -> value) { //inserted
50                   return;
51             } else if (d > new_node -> value) {
52                   if (new_node -> right == NULL) {
53                         new_node -> right = create_tree(d, NULL, NULL);
54                         return;
55                   } else {
56                         new_node = new_node -> right;
57                   }
58             } else { //d < new_node -> value
59                   if (new_node -> left == NULL) {
60                         new_node -> left = create_tree(d, NULL, NULL);
61                         return;
62                   } else {
63                         new_node = new_node -> left;
64                   }
65             }
66       }
67
68 }
```

Main 関数で呼ぶ insert_bst 関数の第一引数は、”&”記号を使う。

```
5 int main(void) {
6   // Build a binary search tree
7   Node *root = NULL;
8   insert_bst(&root, 20);
9   insert_bst(&root, 10);
10  insert_bst(&root, 26);
11  insert_bst(&root, 14);
12  insert_bst(&root, 13);
13  insert_bst(&root,  5);
```

ダブルポインタと&を使うと、insert_bst 関数内で書き換えた内容は main 関数側で反映される。**root は*root に指しながら、実際の root のアドレスにあるデータも指すので、insert_bst 関数内で*root を書き換えると root のアドレスにあるデータを変更できる。

insert_bst 関数の解説：

Line 41 ~ 44：根が存在していなければ根を作る。

Lien 46: 根に指すポインタ new_node を作る。

Line 49 ~ 50：d が既に存在したら(もう d の挿入が終わったら)、何もせずに while ループを出て return する。

Line 51 ~ 57：d が現在のノードの値より大きい ->現在のノードの右に d を挿入する。もし現在のノードが既に右の子を持ったら、その子の位置にポインタ new_node を移動する。無限ループ while(1)を使うことで、挿入が終わらない限り適切な位置を探し続ける。

Line 58 ~ 63：d が現在のノードの値より小さい ->現在のノードの左に d を挿入する。もし現在のノードが既に左の子を持ったら、その子の位置にポインタ new_node を移動する。無限ループ while(1)を使うことで、挿入が終わらない限り適切な位置を探し続ける。

- `void delete_bst(Node **root, int d)`
  - root を根とする 2 分探索木から整数 d を削除する．insert_bst 関数と同様に，第一引数に Node 型ポインタ root のポインタをセットしている．

```
111 void delete_bst(Node **root, int d) {
112     if (*root == NULL) {
113         // The tree is empty, or the value does not exist in the tree
114         return;
115     }
116 
117     if (d < (*root)->value) {
118         // If the value to be deleted is smaller than the value of the current node, it is in the left subtree
119         delete_bst(&(*root)->left, d);
120     } else if (d > (*root)->value) {
121         // If the value to be deleted is greater than the value of the current node, it is in the right subtree
122         delete_bst(&(*root)->right, d);
123     } else {
124         // If the value to be deleted is the value of the current node
125         Node *temp;
126 
```

まず、削除したいノードを再帰方法で探す。 d が現在のノードの値より大きかったらポインタを右へ、小さかったら左へ。見つかったら一時的なノード temp を作って、次の処理を行う。

```
127         // Case 1: Node has less than two children
128         if ((*root)->left == NULL) {
129             temp = *root;
130             *root = (*root)->right;
131             free(temp);
132         } else if ((*root)->right == NULL) {
133             temp = *root;
134             *root = (*root)->left;
135             free(temp);
136         } else {
137             // Case 2: Node has two children
138             // Find the minimum value in the right subtree to replace
139             int minValue = min_bst((*root)->right);
140 
141             // Assign the minimum value to the current node
142             (*root)->value = minValue;
143 
144             // Delete the minimum value from the right subtree
145             delete_bst(&(*root)->right, minValue);
146         }
147     }
148 }
```

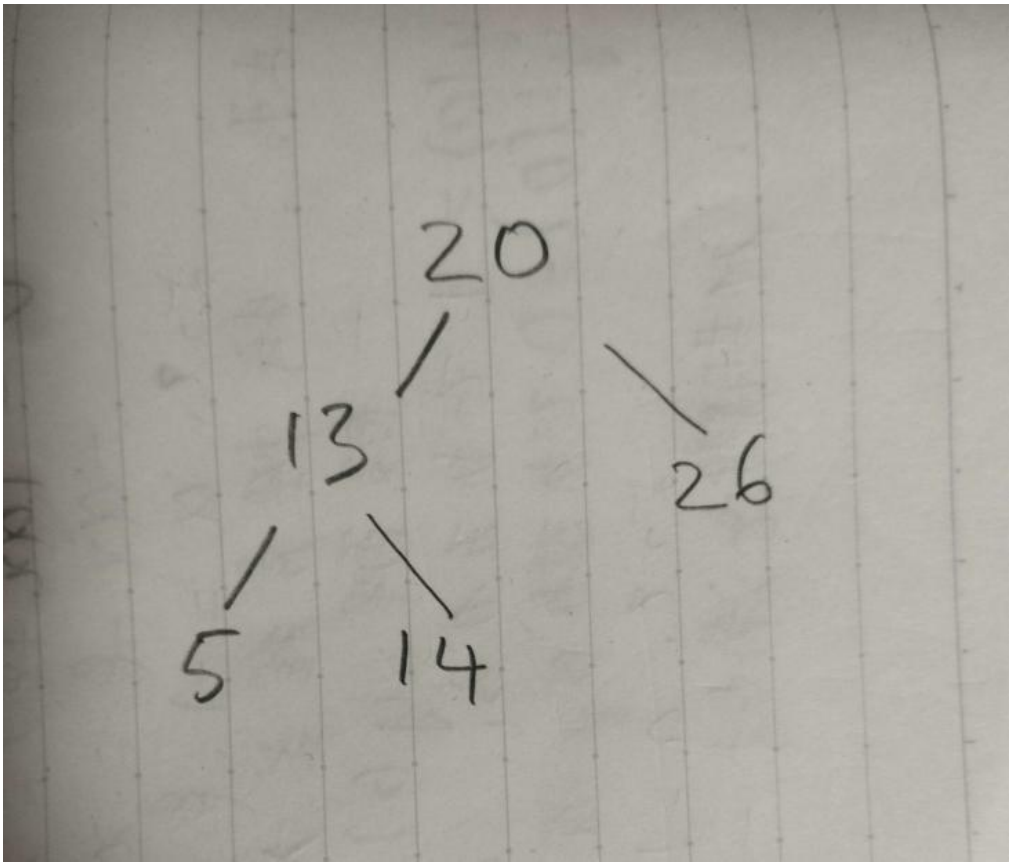
Case 1:削除したいノードが 1 つの子を持つか子を持たないかの場合である。一つの子をもったら、自分を削除して、自分に指すポインタをその子に指すようにする。子を持っていなければ、 1 つの子を持つ時の処理を流用して自分に指すポインタを NULL に指すようにする（Line 130 か Line 134)。その後、自分を削除する（Line 131 か Line 135)。

Case 2: 削除したいノードが 2 つの子を持つ場合：min_bst 関数を使って、右部分木にある最も小さい値を持つノードを探して、そのノードに指すポインタを自分の位置に指すようにしてから、自分を削除する。Line 145 のように再帰を使うことで、削除後の木が回復できる。

4-3-3　実行結果

```
azalea02:~ s2213025$ ./binarysearchtree
inorder: 5 10 13 14 20 26
20(10(5(null,null),14(13(null,null),null)),26(null,null))
Found!: 14
Not found!: 7
Minimum value of this bst: 5
Value 10 deleted
inorder: 5 13 14 20 26
20(13(5(null,null),14(null,null)),26(null,null))
```

授業のスライドの結果とあっているので、実行が成功した。



削除後の木

4-1-4　感想

ダブルポインタを理解するには結構時間がかかった。後 delete_bst 関数も難しかった。

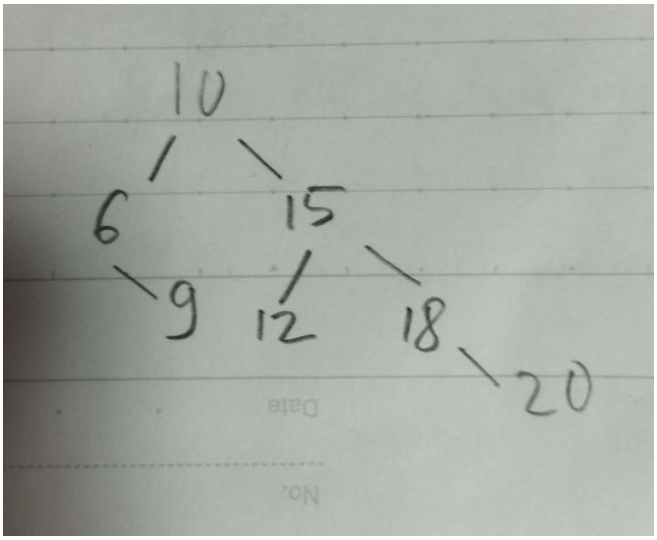
課題 4-4

4-4-1 実装の仕様

- 以下の要件を全て満たすことを確認すること.
  - 空の 2 分探索木に対して，値 10, 15, 18, 6, 12, 20, 9 をこの順で挿入したときの作られる 2 分探索木を確認せよ
  - 以下のすべての場合に対して探索が正しく動作することを確認せよ
    - 探索するデータが根にある場合
    - 葉にある場合
    - 根以外の非終端節点にある場合
    - データが 2 分探索木にない場合
  - 2 分探索木が空の場合，空でない場合それぞれに対して，最小値の探索が正しく動作することを確認せよ
  - 削除のアルゴリズムは複雑なので，以下のようなことを考慮しながら十分にテストし，正しく動作することを確認せよ．また，各テストがどのような場合のテストであるかを明確かつ簡潔に説明すること.
    - 切除される節点が子供を何個持つか
    - 切除される節点が根であるか

4-4-2 実装コードおよび実装コードの説明

まず、main_binarytree.c ファイルに、その木にある節点を作る。また、課題 4-3 に使った節点をコメントにする。



課題の木

- 空の 2 分探索木に対して，値 10, 15, 18, 6, 12, 20, 9 をこの順で挿入したときの作られる 2 分探索木を確認せよ

main 関数を変更する。

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

  /*
  insert_bst(&root, 20);
  insert_bst(&root, 10);
  insert_bst(&root, 26);
  insert_bst(&root, 14);
  insert_bst(&root, 13);
  insert_bst(&root,  5);
  */
  //Excersise 2
  printf("When tree is empty: \n");
  printf("Minimum value of this EMPTY bst: %d\n", min_bst(root));

  insert_bst(&root, 10);
  insert_bst(&root, 15);
  insert_bst(&root, 18);
  insert_bst(&root, 6);
  insert_bst(&root, 12);
  insert_bst(&root, 20);
  insert_bst(&root, 9);
```

- 以下のすべての場合に対して探索が正しく動作することを確認せよ
  - 探索するデータが根にある場合
  - 葉にある場合
  - 根以外の非終端節点にある場合
  - データが 2 分探索木にない場合

Main 関数をさらに変更する：

```
44  //Excersise 2
45     printf("root!\n");
46  if (search_bst(root, 10)) {
47    printf("Found!: 10\n");
48  } else {
49    printf("Not found!: 10\n");
50  }
51
52    printf("leaf!\n");
53  if (search_bst(root, 20)) {
54    printf("Found!: 20\n");
55  } else {
56    printf("Not found!: 20\n");
57  }
58
59    printf("has children!\n");
60  if (search_bst(root, 15)) {
61    printf("Found!: 15\n");
62  } else {
63    printf("Not found!: 15\n");
64  }
```

- 削除のアルゴリズムは複雑なので，以下のようなことを考慮しながら十分にテストし，正しく動作することを確認せよ．また，各テストがどのような場合のテストであるかを明確かつ簡潔に説明すること．
  - 切除される節点が子供を何個持つか
  - 切除される節点が根であるか

main 関数を変更する。

```
82  //excercise 2
83  delete_bst(&root, 10);
84  printf("Value 10 (root) deleted\n");
85
86  printf("inorder: ");
87  inorder(root);
88  printf("\n");
89
90  display(root);
91  printf("\n");
92
93  delete_bst(&root, 15);
94  printf("Value 15 (parent) deleted\n");
95
96  printf("inorder: ");
97  inorder(root);
98  printf("\n");
99
100 display(root);
101 printf("\n");
```

次に実行結果を対照する。

4-4-3　実行結果

```
azalea02:~ s2213025$ ./binarysearchtree
When tree is empty:
Minimum value of this EMPTY bst: -1
inorder: 6 9 10 12 15 18 20
10(6(null,9(null,null)),15(12(null,null),18(null,20(null,null))))
root!
Found!: 10
leaf!
Found!: 20
has children!
Found!: 15
not in the tree!
Not found!: 8
Minimum value of this bst: 6
Value 10 (root) deleted
inorder: 6 9 12 15 18 20
12(6(null,9(null,null)),15(null,18(null,20(null,null))))
Value 15 (parent) deleted
inorder: 6 9 12 18 20
12(6(null,9(null,null)),18(null,20(null,null)))
```

- 空の 2 分探索木に対して，値 10, 15, 18, 6, 12, 20, 9 をこの順で挿入したときの作られる 2 分探索木を確認せよ

```
inorder: 6 9 10 12 15 18 20
10(6(null,9(null,null)),15(12(null,null),18(null,20(null,null))))
```

課題の木とあっている。

- 以下のすべての場合に対して探索が正しく動作することを確認せよ
    - 探索するデータが根にある場合
    - 葉にある場合
    - 根以外の非終端節点にある場合
    - データが 2 分探索木にない場合

```
root!
Found!: 10
leaf!
Found!: 20
has children!
Found!: 15
not in the tree!
Not found!: 8
```

木にある要素が全部見つかったのであっている。

- 2 分探索木が空の場合，空でない場合それぞれに対して，最小値の探索が正しく動作することを確認せよ

```
When tree is empty:
Minimum value of this EMPTY bst: -1
```

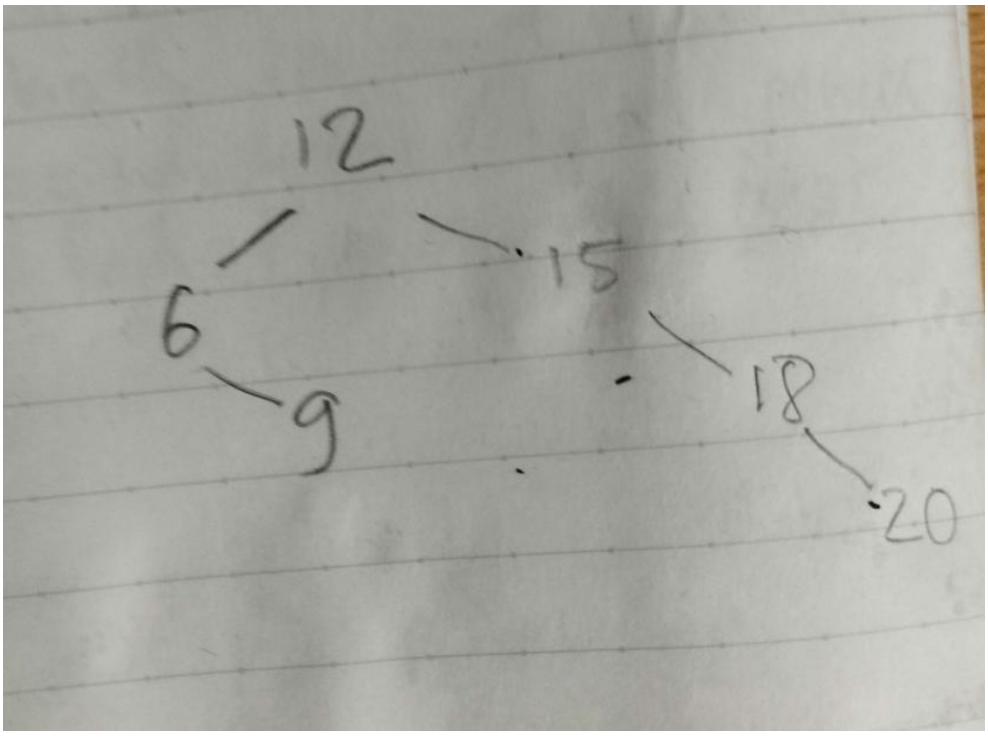
木が空のとき（木を作る前の時に)、min = -1 なので、課題の要求よあっている。

```
Minimum value of this bst: 6
```

気が作った後、min = 6 で、正しいことがわかる。

- 削除のアルゴリズムは複雑なので，以下のようなことを考慮しながら十分にテストし，正しく動作することを確認せよ．また，各テストがどのような場合のテストであるかを明確かつ簡潔に説明すること．
    - 切除される節点が子供を何個持つか
    - 切除される節点が根であるか

まず、根を削除する。木はこうになる。

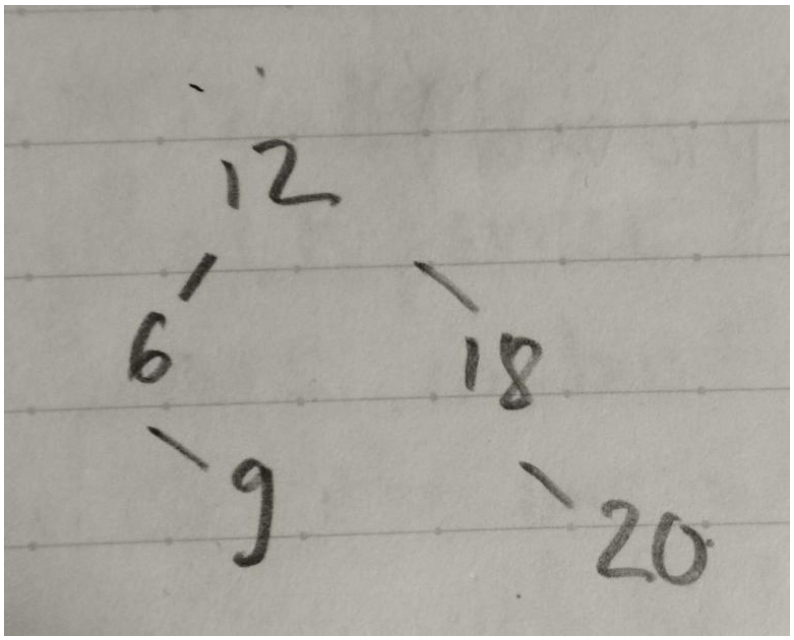


説明：10 が根なので削除される。その後、右部分木の最小値 12 が根になる。

```
Value 10 (root) deleted
inorder: 6 9 12 15 18 20
12(6(null,9(null,null)),15(null,18(null,20(null,null))))
```

見せた木にあっている。

引き続き、子供を何個持つかの節点 15 をさらに削除した後、木はこうになる。




```
Value 15 (parent) deleted
inorder: 6 9 12 18 20
12(6(null,9(null,null)),18(null,20(null,null)))
```

見せた木にあっている。

よって、全部が正しいことがわかる。

1. 発展課題

課題 4-5

4-5-1 実装の仕様

bst_advanced を完成させる。bst_advanced を完成させるためには，insert_bst 関数および delete_bst 関数を bst_advanced 用に実装し直す必要がある (基本課題とは違って，この実装ではそれらの関数は Node 型のポインタを返していることに注意) が，それ以外の関数は基本課題で実装した関数を流用できる．そのため，以下に各関数の説明を示すが，既出のものは割愛する．

- `Node *insert_bst(Node *root, int d)`
    - root を根とする 2 分探索木に整数 d を追加する．2 分探索木が空の場合は，整数 d の根を作成する．**基本課題とは違い，この関数は再帰呼び出しを用いて実装される**．
- `Node *delete_min_bst(Node *root)`
    - root を根とする 2 分探索木における最小値を削除し，その節点が子を持つ場合は木の回復を行った後，その木の根のポインタを返す．
- `Node *delete_bst(Node *root, int d)`
    - root を根とする 2 分探索木から整数 d を削除する．**insert_bst 関数と同様に再帰呼び出しを用いて実装される**．
    - delete_bst 関数の実装では，削除すべき節点の左右の子が両方とも NULL でないときに，min_bst 関数および delete_min_bst 関数を用いる．min_bst 関数により探索した右部分木の最小値を削除対象の節点にコピーし (右部分木の最小値をもつ節点を削除対象の節点の位置に移動させ)，delete_min_bst 関数により最小値をコピーされた節点の削除を行う．

4-5-2 実装コードおよび実装コードの説明

まず、binarytree.c で定義した height 関数を bst_advanced.c に導入する。

```c
int height(Node *n) {
    if (n == NULL) {
        return 0; //empty tree has height = 0, also non-existing node has value 0
    } else {
        int leftHeight = height(n->left);
        int rightHeight = height(n->right);
        // Height of left or right tree, +1 for the root
        return 1 + (leftHeight > rightHeight ? leftHeight : rightHeight);
    }
}
```

導入したら、bst_advanced.h にもこのラインを追加：

```c
int height(Node *n);
```

では、bst_advanced.c の要素を解説する。

- `Node *insert_bst(Node *root, int d)`
    - root を根とする 2 分探索木に整数 d を追加する．2 分探索木が空の場合は，整数 d の根を作成する．**基本課題とは違い，この関数は再帰呼び出しを用いて実装される**．

```c
Node *insert_bst(Node *root, int d) {
    // If the tree is empty, create a new node as the root
    if (root == NULL) {
      Node *new_node = (Node *)malloc(sizeof(Node));
      new_node -> value = d;
      new_node -> left = NULL;
      new_node -> right = NULL;
      new_node -> height = 0;
      return new_node;
   }

    // If the value to insert is smaller than the value at the current node, insert into the left subtree
    if (d < root->value) {
        root->left = insert_bst(root->left, d);
    }
    // If the value to insert is greater than the value at the current node, insert into the right subtree
    else if (d > root->value) {
        root->right = insert_bst(root->right, d);
    }

    // If the value to insert already exists, do nothing

    return root; // Return the tree after insertion
}
```

Line 45 ~ 51：ノードが存在しなければ、malloc 関数を使ってメモリを与えてノードを作る。

Line 54 ~ 61：d を現在のノードの値と比較する。d のほうが大きかったらポインタを右へ移動する。逆に d のほうが小さかったらポインタを左へ。再帰方法で適切な位置を探して挿入する。

- `Node *delete_min_bst(Node *root)`
    - root を根とする 2 分探索木における最小値を削除し，その節点が子を持つ場合は木の回復を行った後，その木の根のポインタを返す．

```
70 Node *delete_min_bst(Node *root) {
71       // Base case: If the tree is empty, return NULL
72     if (root == NULL) {
73         return NULL;
74    }
75
76       int minValue = min_bst(root);
77
78       // If the current node is the one with the minimum value:
79       if (minValue == root -> value){
80             Node *temp = root;
81
82             //that smallest node has no child:
83             if (root -> right == NULL) {
84                   free(root);
85                   root = NULL; //Set the root to NULL after deletion
86             }
87             //that smallest node has a right child:
88             else {
89                   root = root -> right;
90                   free(temp);
91             }
92
93       }
```

まず、min_bst 関数を呼んで最小値を探す(Line 76)。

見つかったら、そのノードが子を持つかどうか確認する(Line 78 ~ 91)。ここで注意してほしいのは、最小値を持つノードには左の子がない。従って、最小値を持つノードに子があったらそれは右の子でしかない。それを前提として、最小値を持つノードを削除したいときに、自分に指すポインタを右の子に指すようにしてから自分を削除する(Line 88 ~ 90)。

```
94      // If the current node is not the one with the minimum value, move to its left child:
95      else {
96              root->left = delete_min_bst(root -> left);
97      }
98      return root;
99  }
```

また、最小値を持つノードがまだ見つからなかったら再帰方法で引き続き探し続ける。

- `Node *delete_bst(Node *root, int d)`
    - root を根とする 2 分探索木から整数 d を削除する．**insert_bst 関数と同様に再帰呼び出しを用いて実装される**．
    - delete_bst 関数の実装では，削除すべき節点の左右の子が両方とも NULL でないときに，min_bst 関数および delete_min_bst 関数を用いる．min_bst 関数により探索した右部分木の最小値を削除対象の節点にコピーし (右部分木の最小値をもつ節点を削除対象の節点の位置に移動させ)，delete_min_bst 関数により最小値をコピーされた節点の削除を行う．

```
101 Node *delete_bst(Node *root, int d) {
102     // Base case: If the tree is empty, return NULL
103     if (root == NULL) {
104         return NULL;
105     }
106 
107     // If the value to be deleted is smaller than the root's value,
108     // it is in the left subtree
109     if (d < root->value) {
110         root->left = delete_bst(root->left, d);
111     }
112     // If the value to be deleted is greater than the root's value,
113     // it is in the right subtree
114     else if (d > root->value) {
115         root->right = delete_bst(root->right, d);
116     }
```

まず、再帰方法で削除したいノードを探す。

```
117     // If the value to be deleted is equal to the root's value, this is the node to be deleted
118     else {
119         // Case 1: Node has less than two children
120         if (root->left == NULL) {
121             Node *temp = root->right;
122             free(root);
123             return temp;
124         } else if (root->right == NULL) {
125             Node *temp = root->left;
126             free(root);
127             return temp;
128       }
129 
130         // Case 2: Node has two children
131         // Find the minimum value in the right subtree to replace
132         root->value = min_bst(root->right);
133 
134         // Delete the minimum value from the right subtree
135         root->right = delete_min_bst(root->right);
136     }
137 
138     return root; // Return the modified root
```

見つかったら、そのノードに子があるかをさらに確認する。一つの子、または子を持たない場合、自分に指すポインタをその子に指すようにして（子を持たない場合は NULL に指すようにして）から自分を削除する。一方、2 つの子を持ったら、右部分木の最も小さいノードを探して自分に代入する(line 132)。その後 delete_min_bst 関数を使って、その最も小さいノードの元の位置を削除する。

Line 135:もし

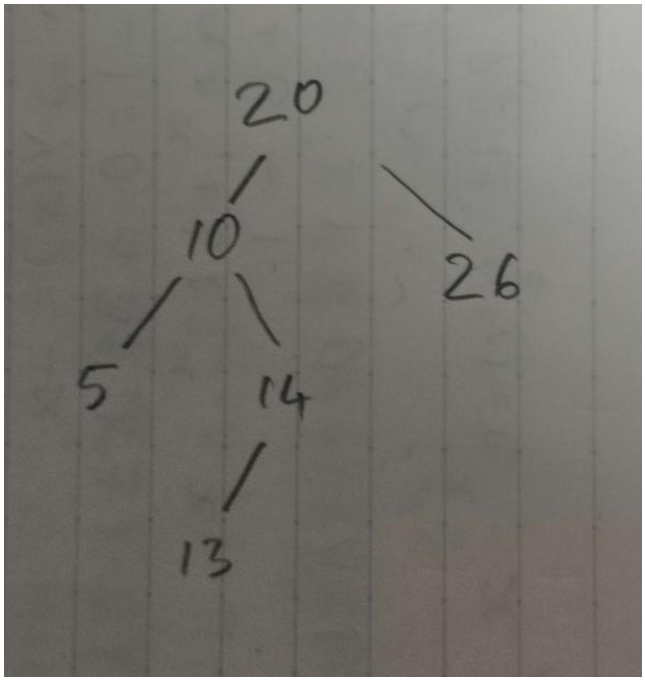
さらに、display 関数もこのように工夫する：

```
161 void display(Node *n) {
162     if (n == NULL) {
163         printf("null");
164         return;
165     }
166
167   n -> height = height(n);
168
169     printf("%d#%d(", n->value, n->height);
170     display(n->left);
171     printf(",");
172     display(n->right);
173     printf(")");
```

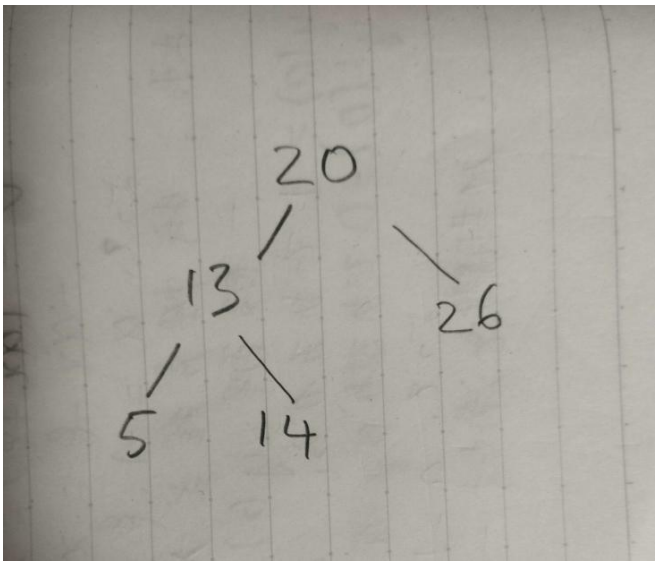
binarytree.c にある関数だが、さらに height 関数を読んで、#記号を加えてノードの高さを表示する。

4-5-3 実行結果

```
azalea02:~ s2213025$ ./bst_advanced
inorder: 5 10 13 14 20 26
20#4(10#3(5#1(null,null),14#2(13#1(null,null),null)),26#1(null,null))
Found!: 14
Not found!: 7
Minimum value of this bst: 5
Value 10 deleted
inorder: 5 13 14 20 26
20#3(13#2(5#1(null,null),14#1(null,null)),26#1(null,null))
```



課題の木



削除後の木

ここで、高さもあっていることがわかる。

4-5-4 感想

delete_bst 関数が思ったよりややこしかった。後は Height 関数も導入するのを注意しないといけない。