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：2023/11/13

連絡事項：ターミナルでソースコードを実行するときの不思議な事項。

実行するときに、こういうエラーが出る可能性がある。

```
main_binarytree.c: In function ‘main’:
main_binarytree.c:47:3: warning: implicit declaration of function ‘create_mirror
’; did you mean ‘create_tree’? [-Wimplicit-function-declaration]
   47 |   create_mirror(mc);
      |   ^~~~~~~~~~~~~
      |   create_tree
main_binarytree.c:52:17: warning: implicit declaration of function ‘are_mirrors’
 [-Wimplicit-function-declaration]
   52 |   bool result = are_mirrors(c, mc);
      |                 ^~~~~~~~~~~
```

これはバグではない。このときに、binarytree.c に好きな波カッコ「{」にクリックして、ctrl S で binarytree.c をもう一度保存したら実行できるはずだ。ただし、binarytree.c を保存したら直ぐにターミナルに移動して実行しなければならない。もし binarytree.c を保存した後 main_binarytree.c に移動したら、上記のエラーがまた出る。

1. 必須課題

課題 4-1

4-1-1 実装の仕様

ラベルとして文字列をとる 2 分木を実現するプログラム binarytree を実装せよ．binarytree.c には，最低限以下の関数を定義すること。

`Node *create_tree(char *label, Node *left, Node *right)`

label を根のラベル，左部分木の根を left，右部分木の根を right とする 2 分木を作成し，作成した 2 分木の根のポインタを返す．

`void preorder(Node *n)`

節点 n を根とする 2 分木を探索し，行きがけ順で節点のラベルを標準出力する．

`void inorder(Node *n)`

節点 n を根とする 2 分木を探索し，通りがけ順で節点のラベルを標準出力する．

`void postorder(Node *n)`

節点 n を根とする 2 分木を探索し，帰りがけ順で節点のラベルを標準出力する．

`void display(Node *n)`

節点 n を根とする 2 分木の構造が完全に分かる形式で標準出力する．例えば，
"C(A(I(null,null),D(F(null,null),null)),L(null,G(null,null)))"はその一例である．

`void breadth_first_search(Node *n)`

節点 n を根とする 2 分木を幅優先探索し，ラベルを標準出力する．課題 2 で実装したキューを上手く活用すること．

`int height(Node *n)`

節点 n を根とする 2 分木の高さを返す．ただし，教科書の定義の高さ + 1 とする．つまり NULL の高さが 0，根のみの木が高さ 1 とすること．

`void delete_tree(Node *n)`

節点 n を根とする 2 分木を削除（2 分木の節点用に確保されたメモリ領域を全て解放）する．

4-1-2 実装コードおよび実装コードの説明

binarytree.c ファイル：

ここで、8 つの関数があるので、1 つずつ開設する。

- `Node *create_tree(char *label, Node *left, Node *right)`
    - label を根のラベル，左部分木の根を left，右部分木の根を right とする 2 分木を作成し，作成した 2 分木の根のポインタを返す．

```
Node *create_tree(char *this_label, Node *this_left, Node *this_right) { //because 3 keywords label, left and right also appear in struct => distinguish
	Node *new_node = (Node *)malloc(sizeof(Node));
	new_node -> label = this_label;
	new_node -> left = this_left;
	new_node -> right = this_right;
	//printf("note %s created!\n", new_node -> label);
	return new_node;
}
```

Malloc 関数でノードにメモリを与える。その後、左と右のノードに指すように this_left と this_right を使って定める。

- `void preorder(Node *n)`
  - 節点 n を根とする 2 分木を探索し，行きがけ順で節点のラベルを標準出力する.

```
void preorder(Node *n) {
	if (n == NULL) {
		return;
	}
	printf("%s ", n -> label);
	preorder(n -> left);
	preorder(n -> right);
}
```

教科書に書いてあるもの通りに定める。

- `void inorder(Node *n)`
  - 節点 n を根とする 2 分木を探索し，通りがけ順で節点のラベルを標準出力する.

```
void inorder(Node *n) { //go from leftmost node to root then go to rightmost node
	if (n == NULL) {
		return;
	}
	inorder(n -> left);
	printf("%s ", n -> label); //1st time: already go to the left most node
	inorder(n -> right);
}
```

教科書に書いてあるもの通りに定める。

- `void postorder(Node *n)`
    - 節点 n を根とする 2 分木を探索し，帰りがけ順で節点のラベルを標準出力する．

```
void postorder(Node *n) {
      if (n == NULL) {
            return;
      }
      postorder(n -> left);
      postorder(n -> right);
      printf("%s ", n -> label);
}
```

教科書に書いてあるもの通りに定める。

- `void breadth_first_search(Node *n)`
    - 節点 n を根とする 2 分木を幅優先探索し，ラベルを標準出力する．課題 2 で実装したキューを上手く活用すること．

まず、キュー、enqueue 関数、dequeue 関数を作る。

```
44 ////////////////////////////////////////
45 //Define enqueue and dequeue function
46
47 typedef struct queue {
48   Node **buffer;
49   int length;
50   int front;
51   int rear;
52 } Queue;
```

```
54 Queue *create_queue(int len){
55       Queue *new_queue = (Queue*)malloc(sizeof(Queue));
56
57       new_queue -> buffer = (Node **)malloc(sizeof(Node *) * len);  //the queue
58       new_queue -> front = 0;
59       new_queue -> rear = 0;
60       new_queue -> length = len;
61
62       printf("%s\n", "Queue created!");
63       return new_queue;
64 }
```

キューに格納するものは Node 型のデータだから、buffer にメモリを与えるための malloc 関数は Node を使う。

```
67 void enqueue(Queue *q, Node *d) {
68       if (q -> rear > q -> length - 1) { //rear => push
69             q -> rear = 0; //make a ring buffer
70       }
71
72       q -> buffer[q -> rear] = d;
73       q -> rear += 1;
74
77 Node *dequeue(Queue *q){
78       if (q -> front > q -> length - 1) { //front => pop
79             q -> front = 0; //make a ring buffer
80 }
81       Node *x = q -> buffer[q -> front];
82       q -> front += 1;
83
84       return x;
85 }
86
```

注意してほしいのは dequeue 関数である。返す価値が Node 型にしなければならない。その理由は breadth_first_search 関数で解説する。

breadth_first_search

```
90 void breadth_first_search(Node *n) {
91       Queue *q = create_queue(20); //Size of queue
92       enqueue (q, n);
93       while ((q -> rear) > (q -> front)) { //while there is value(s) in queue
94             Node *current_n = dequeue (q);
95             printf("%s ", current_n -> label);
96             if (current_n -> left != NULL) {
97                   enqueue (q, current_n -> left);
98             }
99             if (current_n -> right != NULL) {
100                  enqueue (q, current_n -> right);
101            }
102      }
103
104      free(q -> buffer);
105      free(q);
106      printf("Queue deleted!\n ");
107 }
```

Line 94: Dequeue されたものは節点の label ではなく、節点にさすポインタであるのを注意してほしい。そのため、上記の dequeue 関数が返す価値を Node 型に設定しなければならない。

Line 96 & line 99: Dequeue された節点が子を持ったら、その子をまた Enqueue する。

教科書に書いてあるアルゴリズムとほとんど同じであるから、わかりにくいものではないはずだ。

Line 104 & 105: キューに与えたメモリを開放する。

Line 106: キューのメモリをちゃんと解放したか確認するため。

- `void display(Node *n)`
    - 節点 n を根とする 2 分木の構造が完全に分かる形式で標準出力する．例えば，"C(A(I(null,null),D(F(null,null),null)),L(null,G(null,null)))"はその一例である．

```
108 void display(Node *n) {
109     if (n == NULL) {
110         printf("null");
111         return;
112     }
113 
114     printf("%s(", n->label);
115     display(n->left);
116     printf(",");
117     display(n->right);
118     printf(")");
119 
120 }
```

再帰処理を使う。まずは n のラベルをプリントする。その後、左の子をプリントしてから、コンマで区切って右の子をプリントする。存在しないものは”null”とプリントする。

- `int height(Node *n)`
    - 節点 n を根とする 2 分木の高さを返す．ただし，教科書の定義の高さ + 1 とする．つまり NULL の高さが 0，根のみの木が高さ 1 とすること．

```
122 int height(Node *n) {
123     if (n == NULL) {
124         return 0; //empty tree has height = 0, also non-existing node has value 0
125     } else {
126         int leftHeight = height(n->left);
127         int rightHeight = height(n->right);
128         // Height of left or right tree, +1 for the root
129         return 1 + (leftHeight > rightHeight ? leftHeight : rightHeight);
130     }
131 }
```

面白い関数である。まず root がら始まる。

Line 123 ~ 124:子を持たなければ 0 を返す。

Line 129: 子を持ったら 1 を足す。

面白いというのは、ポインタが各々節点の存在しない子まで移動することだ。Line 126 と line 127 は、木の葉に移動したら、その葉が持っていない子に移動して、何も存在しないと気づいて、0 を返す。

Line 129:root の右の高さと左の高さを比較して、より大きいものを使う。root が 1 であるから、最後的に 1 を足さなければならない。

- `void delete_tree(Node *n)`
    - 節点 n を根とする 2 分木を削除 (2 分木の節点用に確保されたメモリ領域を全て解放) する.

```
168 void delete_tree(Node *n) {
169     if (n == NULL) {
170         return; // empty tree or current node is NULL
171       }
172
173       delete_tree(n -> left);
174       delete_tree(n -> right);
175       free(n);
176 }
```

与えたメモリを開放する。注意してほしいのは、ある節点 n を消したいときに、先にその節点の持つ子を消さなければならない。逆にすると segmentation fault になる。

4-1-3　実行結果

```
azalea02:~ s2213025$ ./binarytree
preorder: C A I D F L G
inorder: I A F D C L G
postorder: I F D A G L C
bfs:
Queue created!
C A L I D G F
Queue deleted!

C(A(I(null,null),D(F(null,null),null)),L(null,G(null,null)))
height: 4
```

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。さらに、キューメモリを管理するために queue created! や queue deleted! を加えた。

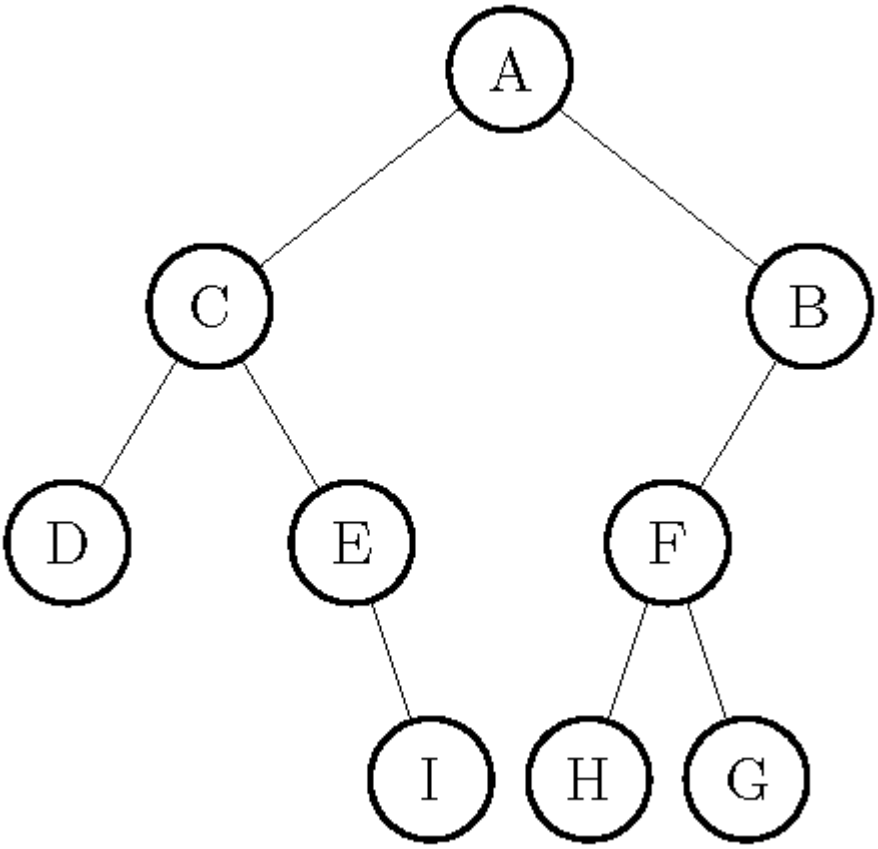
4-1-4　感想

`breadth_first_search` 関数が一番ややこしかった。なぜかというと、キューを作らなければならないからだ。それに、dequeue 関数を工夫する必要がある。

課題 4-2

4-2-1 実装の仕様

以下の 2 分木に対して，1 で゛実装した関数が゛正しく動作することを確認すること.



4-2-2 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 4-1 に使った節点をコメントにする。

```
7  int main(void) {
8    // Build a binary tree
9    /*
10   Node *f = create_tree("F", NULL, NULL);
11   Node *i = create_tree("I", NULL, NULL);
12   Node *d = create_tree("D", f, NULL);
13   Node *g = create_tree("G", NULL, NULL);
14   Node *a = create_tree("A", i, d);
15   Node *l = create_tree("L", NULL, g);
16   Node *c = create_tree("C", a, l);
17   */
18
19   printf("Ex 4-2\n");
20   Node *i = create_tree("I", NULL, NULL);
21   Node *e = create_tree("E", NULL, i);
22   Node *d = create_tree("D", NULL, NULL);
23   Node *c = create_tree("C", d, e);
24   Node *g = create_tree("G", NULL, NULL);
25   Node *h = create_tree("H", NULL, NULL);
26   Node *f = create_tree("F", h, g);
27   Node *b = create_tree("B", f, NULL);
28   Node *a = create_tree("A", c, b);
```

それに、preorder、inorder、postorder、breadth_first_search、display, delete 関数には変数 a を渡す。

```
31  printf("preorder: ");
32  preorder(a);
33  printf("\n");
34
35  printf("inorder: ");
36  inorder(a);
37  printf("\n");
38
39  printf("postorder: ");
40  postorder(a);
41  printf("\n");
42
43  printf("bfs: ");
44  breadth_first_search(a);
45  printf("\n");
46
47  display(a);
48  printf("\n");
49  printf("height: %d\n", height(a));

72  delete_tree(a);
```

4-2-3　実行結果

```
azalea02:~ s2213025$ ./binarytree
Ex 4-2
preorder: A C D E I B F H G
inorder: D C E I A H F G B
postorder: D I E C H G F B A
bfs:
Queue created!
A C B D E F I H G
Queue deleted!

A(C(D(null,null),E(null,I(null,null))),B(F(H(null,null),G(null,null)),null))
height: 4
```

全部当っていることがわかる。

2. 発展課題

課題 4-3

4-3-1 実装の仕様

基本課題で実装した binarytree に以下の機能を有する関数を追加し，正しく動作することを確認することを確認すること．

- `Node *create_mirror(Node *n)`
    - 節点 n を根とする 2 分木の鏡像を作成し，作成した鏡像の根のポインタを返す．2 分木の鏡像とは全ての非葉ノードの左右の子が入れ替わった木のことであり，下に示す Mirror は Original の鏡像である．
- `bool are_mirrors(Node *n0, Node *n1)`
    - 節点 n0 を根とする 2 分木と節点 n1 を根とする 2 分木が互いに鏡像の関係になっていれば true を，そうでなければ false を bool 型で返す．例えば，下に示す Original の根のポインタと Mirror の根のポインタを引数として渡せば，true が出力される．

4-3-2 実装コードおよび実装コードの説明

Binarytree.c

- `Node *create_mirror(Node *n)`

節点 n を根とする 2 分木の鏡像を作成し，作成した鏡像の根のポインタを返す．2 分木の鏡像とは全ての非葉ノードの左右の子が入れ替わった木のことであり，下に示す Mirror は Original の鏡像である

```
134 Node *create_mirror(Node *n) {
135         Node *temp;
136         if (n == NULL) {
137                 return n;
138         } else {
139                 create_mirror(n -> left);
140                 create_mirror(n -> right);
141                 temp = n -> left;
142                 n -> left = n -> right;
143                 n -> right = temp;
144         }
145 }
```

Height 関数とほとんど同じである。各々節点に移動して、右の子と左の子を入れ替える。注意してほしいのは、葉の持っていない右の子と左の子も入れ替える。だたそれは null と null を入れ替えるから問題ない。

- `bool are_mirrors(Node *n0, Node *n1)`
    - 節点 n0 を根とする 2 分木と節点 n1 を根とする 2 分木が互いに鏡像の関係になっていれば true を，そうでなければ false を bool 型で返す．例えば，下に示す Original の根のポインタと Mirror の根のポインタを引数として渡せば，true が出力される．

```
148 bool are_mirrors(Node *n0, Node *n1) {
149     // leaf node
150     if (n0 == NULL && n1 == NULL) {
151         return true;
152     }
153
154     // One node is NULL, the other is not (both nodes are not leaf node, because leaf nodes have already been
    checked recently)
155     if (n0 == NULL || n1 == NULL) {
156         return false;
157     }
158
159     // check label
160     if (strcmp(n0->label, n1->label) != 0) {
161         return false;
162     }
163
164     // Both are true => return true
165     return are_mirrors(n0->left, n1->right) && are_mirrors(n0->right, n1->left);
166 }
167
```

Line 150 ~ 152:　両方が葉であり、true を返す。

Line 155 ~ 157:　既に葉を確認したから、また子が存在しない節点が見つかったら、false を返す。

Line 159: strcmp 関数を使って、両方の節点の label を対照する。違ったら false を返す。

Line 165: AND ゲートみたいに、２つの節点が持つ右の子と左の子が互いに鏡像の関係になっていれば true。

main_binarytree.c

まず、課題の期と全く同じ木を作る。だた、節点のポインタをm(mirror)を加えて命名する。

```
52  printf ("\nStart creating mirror!\n");
53  //create mirror nodes
54  Node *mf = create_tree("F", NULL, NULL);
55  Node *mi = create_tree("I", NULL, NULL);
56  Node *md = create_tree("D", mf, NULL);
57  Node *mg = create_tree("G", NULL, NULL);
58  Node *ma = create_tree("A", mi, md);
59  Node *ml = create_tree("L", NULL, mg);
60  Node *mc = create_tree("C", ma, ml);
```

鏡を作って表示する。その後、互いに鏡像の関係になっていたかどうか are_mirror 関数使う。

```
62  create_mirror(mc);
63
64  display(mc);
65  printf("\n");
66
67  bool result = are_mirrors(c, mc);
68  printf(result ? "true" : "false");
69  printf("\n");
```

最後、2 つの木を消す。

```
72   delete_tree(c);
73   delete_tree(mc);
```

4-3-3　実行結果

```
C(A(I(null,null),D(F(null,null),null)),L(null,G(null,null)))
height: 4

Start creating mirror!
C(L(G(null,null),null),A(D(null,F(null,null)),I(null,null)))
true
```

Height の上は元の木、Start creating mirror の下は鏡の木。あっていることがわかる。

4-3-4 感想

are_mirrors が非常に難しかった。逆に creat_mirror が思ったより楽だった。ただし、連絡事項に書いている不思議な事項がよく出るので注意しないといけない。

Update: “Node *create_mirror(Node *n);”を binarytree.h に追加すれば(下の写真の Line 18)、そのエラーが解決できる。

```c
#ifndef INCLUDE_GUARD_BINARYTREE_H
#define INCLUDE_GUARD_BINARYTREE_H

typedef struct node {
  char *label;
  struct node *left;
  struct node *right;
} Node;

Node *create_tree(char *label, Node *left, Node *right);
void preorder(Node *n);
void inorder(Node *n);
void postorder(Node *n);
void display(Node *n);
void breadth_first_search(Node *n);
int height(Node *n);
void delete_tree(Node *n);
Node *create_mirror(Node *n); //added
bool are_mirrors(Node *n0, Node *n1);
#endif  // INCLUDE_GUARD_BINARYTREE_H
```