

Data Structures

**Assignment 05**

**Technical Report**

2176211 Jiwon Yang

## I. Problem 2

### 1. Code Explanation

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct TreeNode {
5      int data;
6      struct TreeNode *left, *right, *parent;
7  } TreeNode;
8
9  /*
10     |       |       C       G       F
11     |       |       A   B       D   E
12
13  TreeNode n1 = {'A', NULL, NULL, NULL};
14  TreeNode n2 = {'B', NULL, NULL, NULL};
15  TreeNode n3 = {'C', &n1, &n2, NULL};
16  TreeNode n4 = {'D', NULL, NULL, NULL};
17  TreeNode n5 = {'E', NULL, NULL, NULL};
18  TreeNode n6 = {'F', &n4, &n5, NULL};
19  TreeNode n7 = {'G', &n3, &n6, NULL};
20  TreeNode *exp = &n7;
21
```

1. Struct TreeNode has int type data, and TreeNode pointer type left, right, and parent. Each element indicates the node's left child, right child, and parent's address).
2. Create instances of the TreeNode struct and initialize them with data and pointers to their left, right, and parent nodes, based on the tree structure provided in the comment.

```
22  //find node p's successor
23  //1. if node p has right subtree -> find leftmost node
24  //2. if node p does not have right subtree -> find parent
25  TreeNode *tree_successor(TreeNode *p){
26
27
28      // 오른쪽 서브트리의 가장 왼쪽 노드 찾아서 반환하기.
29      if (p->right != NULL){
30          TreeNode *temp = p->right; //지금 찾기 시작하는 오른쪽 서브트리의 루트
31          while (temp->left != NULL){
32              temp = temp->left;
33          }
34          return temp;
35      }
36
37      else { //오른쪽 서브트리가 비지 않았으면, 현 p의 부모를 temp로 잡기.
38          //부모를 찾을 건데, 어떤 부모냐면 '나'랑 '부모'가 왼쪽모녀 관계일때 멈춤.
39
40          TreeNode *temp = p->parent;
41          while (temp != NULL && temp->right == p){ //오른쪽모녀 관계인 한 계속 반복해야.
42              p = temp;
43              temp = temp->parent;
44          }
45          return temp;
46      }
47  }
48
```

The `tree_successor` function, which takes `TreeNode p` as an argument, is used to find and return the successor of `p`.

1. If `p` has a right subtree, it searches for the leftmost node in that right subtree to find the successor. To achieve this, it employs a temporary variable called `temp`. Initially, `temp` is set to the root of `p`'s right subtree. It continues to move to the left in the right subtree until it encounters a node where the left child is `NULL`. When this condition is met, the function returns the `temp` value, which represents the leftmost node in `p`'s right subtree, making it `p`'s successor.
2. If `p` does not have a right subtree, its successor is found by moving up the tree level by level, searching for the node where `p` is the left child of its parent. In this case, the temporary variable `temp` initially points to `p`'s parent node. The loop continues as long as the condition is satisfied, which means that `temp` is a parent node, and its child relationship with that parent is on the left side. When this condition is no longer met, the loop exits, and the function returns the value of `temp`, which represents the parent of `p`. This parent node is the successor of `p`.

```
49  int main(){
50
51      TreeNode *q = exp;
52      n1.parent = &n3;
53      n2.parent = &n3;
54      n3.parent = &n7;
55      n4.parent = &n6;
56      n5.parent = &n6;
57      n6.parent = &n7;
58      n7.parent = NULL;
59
60      while (q->left) q = q->left;
61      do {
62          printf("%c\n", q->data);
63          q = tree_successor(q);
64      } while (q);
65
66      return 0;
67  }
```

1. After setting the parent node for each node, make node `q` point to the root node. Then, position node `q` to point to the leftmost node in the tree, and move it one by one to its successor to print the tree in inorder traversal.

## 2. Experimental Result

(1) Expected Result :

(2) Actual Result :

Output:

A  
C  
B  
G  
D  
F  
E

A  
C  
B  
G  
D  
F  
E

## 3. Result Analysis

The `tree_successor` function is used in the do-while loop in the main function to visit all nodes in the tree. The time complexity of the `tree_successor` function varies depending on the given node `p`. If the node `p` has a right subtree, it needs to find the leftmost node in that subtree, and this operation takes time proportional to the height of the right subtree. In general, in a binary tree, this operation typically takes  $O(\log n)$  time.

However, if the node `p` does not have a right subtree and needs to move to its parent, the time it takes to move to the parent is proportional to the depth of the tree. In the worst case, this can take  $O(n)$  time, as it might involve traversing the entire tree.

Thus, the time complexity of the `tree_successor` function depends on the structure of the given tree. In the worst case, it can be  $O(n)$ , while in the best case, it's  $O(\log n)$  due to the properties of balanced binary trees.

## II. Problem 3

### 1. Code Explanation

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct TreeNode {
5      int data;
6      struct TreeNode *left, *right, *parent;
7  } TreeNode;
8
9  /*
10  |           G
11  |   C       F
12  |  / \     / \
13  | A  B   D  E
14  */
15
16  TreeNode n1 = {'A', NULL, NULL, NULL};
17  TreeNode n2 = {'B', NULL, NULL, NULL};
18  TreeNode n3 = {'C', &n1, &n2, NULL};
19  TreeNode n4 = {'D', NULL, NULL, NULL};
20  TreeNode n5 = {'E', NULL, NULL, NULL};
21  TreeNode n6 = {'F', &n4, &n5, NULL};
22  TreeNode n7 = {'G', &n3, &n6, NULL};
23  TreeNode *exp = &n7;
```

Same as Problem 1.

```
22  // find node p's predecessor
23  //1. if node p has left subtree, find rightmost node.
24  //2. if node p does not have left subtree, -> find parent.
25  TreeNode *tree_predecessor(TreeNode *p){
26
27      if (p->left != NULL){
28          //left subtree의 가장 오른쪽 노드 찾기.
29          TreeNode *temp = p->left;
30          while (temp->right != NULL){
31              temp = temp->right;
32          }
33          return temp;
34      }
35
36      else {
37          // 내가 오른쪽 자식인 엄마 찾기.
38          // 즉, 엄마가 낳기 전 & 모녀가 왼쪽인 한 계속 뚫
39          TreeNode *temp = p->parent;
40          while (temp != NULL && temp->left == p){
41              p = temp;
42              temp = temp->parent;
43          }
44          return temp;
45      }
46  }
```

The "tree\_predecessor" function finds and returns the predecessor of the given TreeNode "p."

Here are the two scenarios for determining the predecessor:

1. If the given "p" has a left subtree, its predecessor is the rightmost node in that left subtree. To find this node, we declare a temporary variable "temp" and initialize it with the root node value of "p's" left subtree. Then, we continue moving "temp" to the right child's position until there are no more nodes to the right (i.e., until "temp" represents the rightmost node). When the loop exits, the value returned by "temp" becomes the predecessor of "p."
2. If the given "p" does not have a left subtree, its predecessor is found by moving up the tree to the parent nodes. In this case, the predecessor is the parent node where the relationship between the current node and its parent is a rightward connection. To implement this, we initialize a temporary variable "temp" with "p's" parent node. While "temp" is not NULL and its relationship with its child is on the left side, we continue moving up the tree by assigning "temp" to its parent. When the condition is no longer met, the "temp" variable, at that point, represents the first node that was the predecessor of "p."

```
50  int main(){
51
52      TreeNode *q = exp;
53      n1.parent = &n3;
54      n2.parent = &n3;
55      n3.parent = &n7;
56      n4.parent = &n6;
57      n5.parent = &n6;
58      n6.parent = &n7;
59      n7.parent = NULL;
60
61      //go to the rightmost node
62      while (q->right) q = q->right;
63
64
65      //print in reverse order
66      do {
67          printf("%c\n", q->data);
68          q = tree_predecessor(q);
69      } while (q);
70
71      return 0;
72  }
```


1. Unlike in problem 2, in this case, we start from the rightmost node of the tree and systematically traverse the nodes in reverse order by moving from one node to its predecessor using the "tree\_predecessor" function.

## 2. Experimental Result

(1) Expected Result :

(2) Actual Result :

Output:
E
F
D
G
B
C
A



## 3. Result Analysis

The time complexity, when using the "tree\_predecessor" function, remains the same, with a maximum of  $O(n)$  and a minimum of  $O(\log n)$  required.

### III. Problem 4

## 1. Code Explanation

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct TreeNode {
5      int key;
6      struct TreeNode *left, *right;
7  } TreeNode;
8
9  /*
10     35
11     /  \
12    18   68
13   /  \  /  \
14  7   26 99
15 /  \
16 3   12 22 30
17 /  \
18 10  24
19 */
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

```
28 void inorder(TreeNode *root){
29     if (root != NULL) {
30         inorder(root->left);
31         printf("%d\t", root->key);
32         inorder(root->right); }
33 }
```

1. This function, `inorder`, performs an inorder traversal of a binary tree. It takes the root node of the tree as an argument and prints the nodes in ascending order (sorted) by recursively visiting the left subtree, printing the current node's key, and then recursively visiting the right subtree.



```

35 //1. find node with key (which will be deleted)
36 //2. 삭제할 노드 t, 그리고 그 노드의 부모 p가 필요함. (그래야 삭제 가능)
37
38 //3-1. t가 leaf node인 경우 --> 부모 연결 끊으면 끝.
39 //3-2. t가 한쪽 연결 가진 경우 --> 그 subtree를 붙이면 끝.
40 //3-3. t가 양쪽 연결 가진 경우
41 //--> 왼쪽 서브트리의 rightmost (=predecessor) 찾아서 삽입 && 그의 아래 서브트리를 attach.
42 void delete_node (TreeNode **root, int key){
43     TreeNode *t, *p;
44     TreeNode *child, *predec, *predec_p;
45     p = NULL;
46     t = *root;
47
48     //먼저 삭제할 노드 찾기. t랑 p.
49     while (t != NULL && t->key != key){
50         p = t;
51         if (key > t->key) {
52             t = t->right;
53         } else {t = t->left;}
54     }
55
56     //찾았는데 해당 노드가 없던 경우.
57     if (t == NULL){
58         printf("key is not in the tree");
59         return;
60     }
61

```

This function, `delete_node`, is responsible for deleting a node with a given key from a binary search tree. It first searches for the node with the specified key (`key`) and maintains the parent node (`p`). It then handles three different cases of deletion:

```

62 //case 1
63 if (t->left == NULL && t->right == NULL){
64
65     if (p != NULL){
66         if (p->left == t) p->left = NULL;
67         else p->right = NULL;
68     }
69
70     else { //부모 없음 = 삭제하려던 노드가 루트.
71         *root = NULL;
72     }
73 }
74
75 //case 2
76 //두개다 null인 경우는 위에서 이미 함. (leaf node)
77 else if (t->left == NULL || t->right == NULL) {
78
79     if (t->left == NULL) child = t->right;
80     else child = t->left;
81
82     if (p != NULL){
83         if (p->left == t) p->left = child;
84         else p->right = child;
85     }
86
87     else {
88         *root = NULL;
89     }
90 }
91

```

1. Case 1: If the node to be deleted (t) is a leaf node (has no children), it is simply removed by updating the parent's pointer.
2. Case 2: If the node to be deleted has only one child, it is removed by linking the parent to the single child node.

```

92 //case 3
93 //left와 right 둘 다 있는 경우. 우리는 left subtree의 rightmost node 넣을거임.
94 else { //지금 삭제할 노드 t, 개의 부모 p.
95     predec_p = t;
96     predec = t->left; //left subtree의 root.
97
98     while (predec->right != NULL) { //rightmost predecessor 찾음.
99         predec_p = predec;
100         predec = predec->right;
101     }
102
103     if (predec_p->left == predec)
104         predec_p->left = predec->left;
105     else
106         predec_p->right = predec->left;
107
108     t->key = predec->key;
109     t = predec;
110 }
111
112 free(t);
113
114
115 }

```

3. Case 3: If the node to be deleted has two children, it is more complex. The code finds the predecessor node (predec) which is the maximum value node in the left subtree of the node to be deleted. The key of the predecessor node is copied to the node to be deleted, and then the predecessor node is deleted.

```

117 int main() {
118
119     TreeNode *root = &n1;
120     int key1 = 18;
121     int key2 = 35;
122     int key3 = 7;
123
124     printf("Binary Tree\n");
125     inorder(root);
126     printf("\n\n");
127
128     delete_node(&root, key3);
129
130     printf("\nBinary Tree\n");
131     inorder(root);
132     printf("\n\n");
133
134     return 0;
135 }
136

```

## 2. Experimental Result

(1) Deleting node with key = 18

```
117 int main() {
118
119     TreeNode *root = &n1;
120     int key1 = 18;
121     int key2 = 35;
122     int key3 = 7;
123
124     printf("Binary Tree\n");
125     inorder(root);
126     printf("\n\n");
127
128     delete_node(&root, key1;
129
130     printf("\nBinary Tree\n");
131     inorder(root);
132     printf("\n\n");
133
134     return 0;
135 }
```

Binary Tree											
3	7	10	12	18	22	24	26	30	35	68	99
Binary Tree											
3	7	10	12	22	24	26	30	35	68	99	

(2) Deleting node with key = 35

```
117 int main() {
118
119     TreeNode *root = &n1;
120     int key1 = 18;
121     int key2 = 35;
122     int key3 = 7;
123
124     printf("Binary Tree\n");
125     inorder(root);
126     printf("\n\n");
127
128     delete_node(&root, key2;
129
130     printf("\nBinary Tree\n");
131     inorder(root);
132     printf("\n\n");
133
134     return 0;
135 }
```

Binary Tree											
3	7	10	12	18	22	24	26	30	35	68	99
Binary Tree											
3	7	10	12	18	22	24	26	30	68	99	

(3) Deleting node with key = 7

```
117 int main() {
118
119     TreeNode *root = &n1;
120     int key1 = 18;
121     int key2 = 35;
122     int key3 = 7;
123
124     printf("Binary Tree\n");
125     inorder(root);
126     printf("\n\n");
127
128     delete_node(&root, key3;
129
130     printf("\nBinary Tree\n");
131     inorder(root);
132     printf("\n\n");
133
134     return 0;
135 }
```

Binary Tree											
3	7	10	12	18	22	24	26	30	35	68	99
Binary Tree											
3	10	12	18	22	24	26	30	35	68	99	

### 3. Result Analysis

Delete\_node function, which deletes the delivered parameter node  $p$  in the tree, first searches the position of node  $p$ .

When searching for the node to delete, in the worst-case, the function traverses the binary search tree from the root to the node to be deleted. The time complexity of this is  $O(h)$ , where  $h$  is the height of the binary search tree.

When handling three different deletion cases, the worst time complexity would take  $O(h)$  time, in the case3 with traversing the left subtree of the node to be deleted.

Thus, in the best case, the height of the tree is  $O(\log n)$ , making the time complexity  $O(\log n)$ . In the worst case, the height of the tree can be  $O(n)$ , leading to a worst-case time complexity of  $O(n)$ .