

[Explore Problems](#)[Interview](#)[Contest](#)[Discuss](#)[Store](#)

0

[Description](#)[Solution](#)[Discuss \(765\)](#)[Submissions](#)

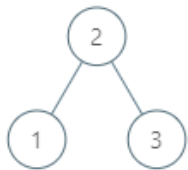
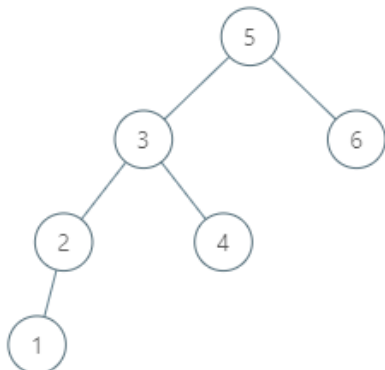
285. Inorder Successor in BST

Medium

[2289](#) [82](#) [Add to List](#) [Share](#)

Given the *root* of a binary search tree and a node *p* in it, return *the in-order successor of that node in the BST*. If the given node has no in-order successor in the tree, return *null*.

The successor of a node *p* is the node with the smallest key greater than *p.val*.

Example 1:**Input:** `root = [2,1,3], p = 1`**Output:** `2`**Explanation:** 1's in-order successor node is 2. Note that both *p* and the return value is of *TreeNode* type.**Example 2:****Input:** `root = [5,3,6,2,4,null,null,1], p = 6`**Output:** `null`**Explanation:** There is no in-order successor of the current node, so the answer is *null*.

Constraints:

- The number of nodes in the tree is in the range $[1, 10^4]$.
- $-10^5 \leq \text{Node.val} \leq 10^5$
- All Nodes will have unique values.


Accepted

286.6K

Submissions

594.6K

Seen this question in a real interview before?

Companies  *i*

v

Related Topics

v

[Tree](#)
[Depth-First Search](#)
[Binary Search Tree](#)
[Binary Tree](#)

Similar Questions

v

[Binary Tree Inorder Traversal](#)

Easy

[Binary Search Tree Iterator](#)

Medium

[Inorder Successor in BST II](#)

Medium

Quick Navigation

★

★

★

★

★

Average Rating: 4.44 (63 votes)

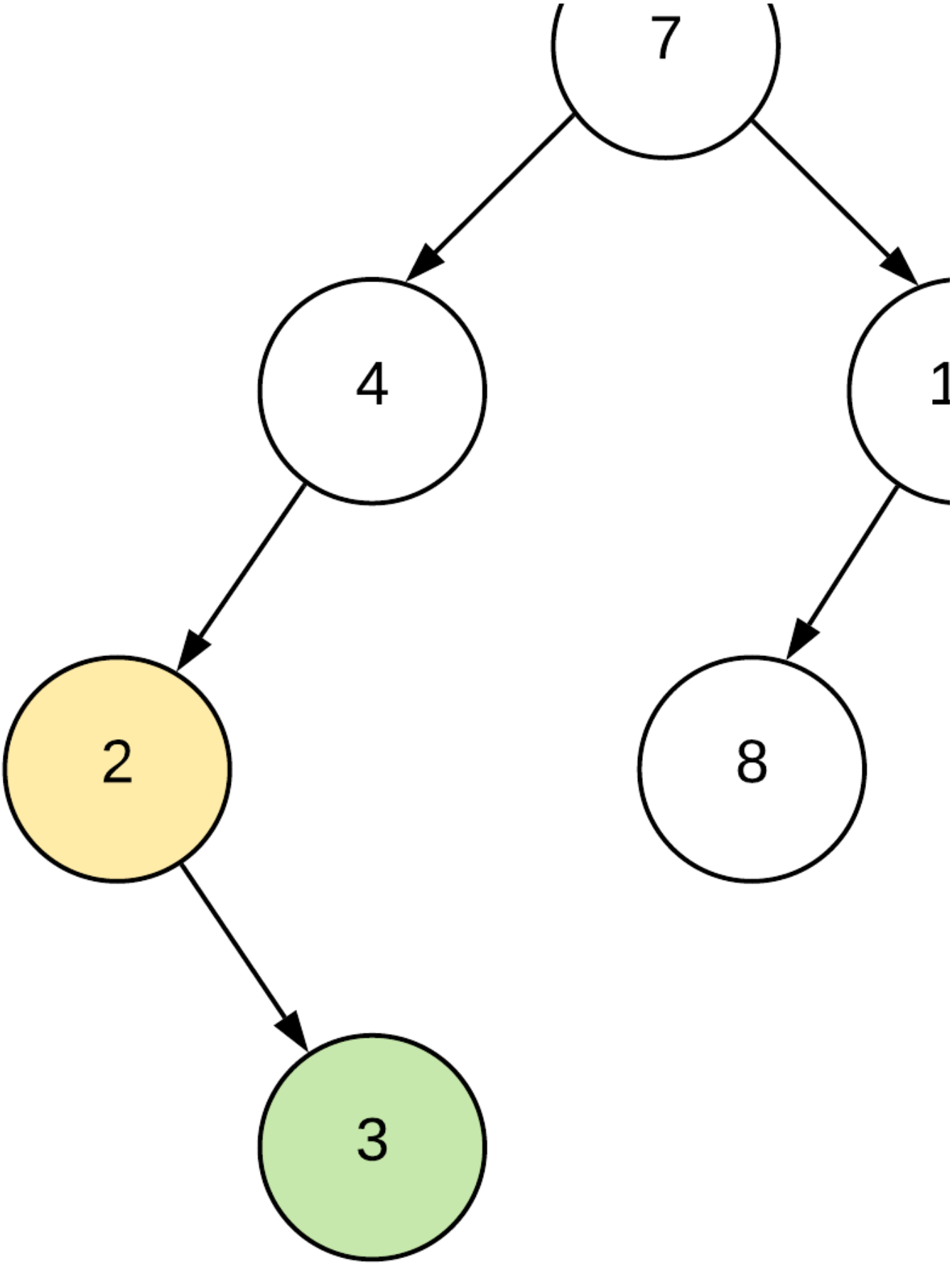
Premium

Solution

Overview

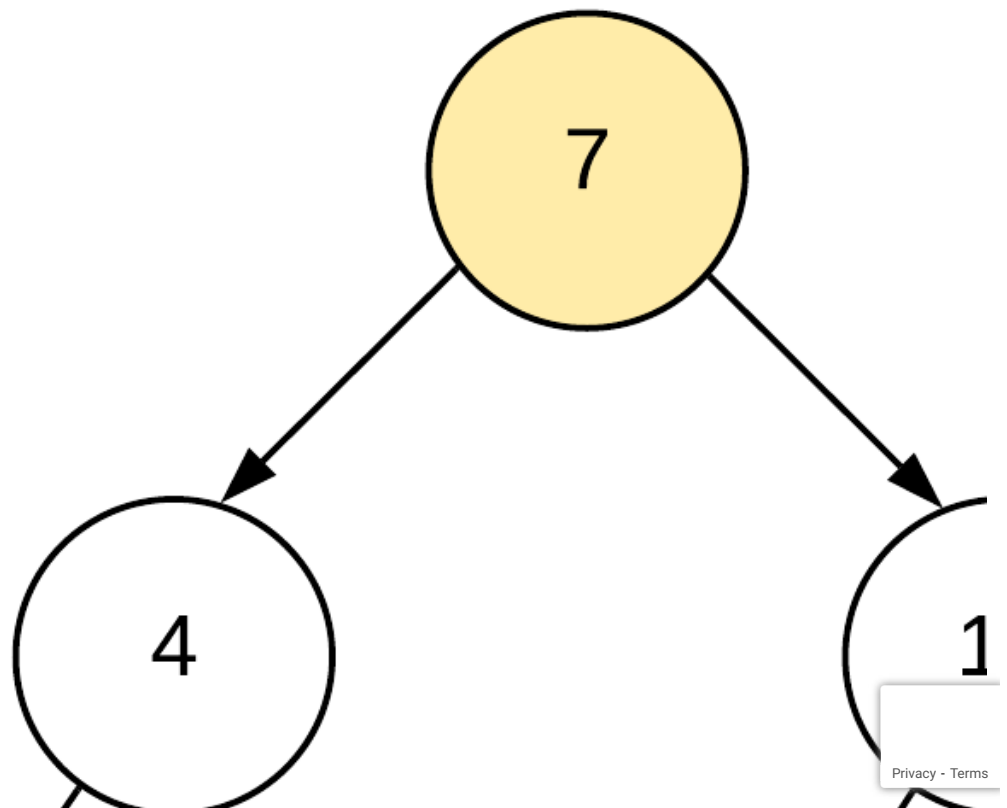
This is a very popular programming interview problem and there are a couple of ways we can approach it. This problem is very similar to finding the Inorder Successor in a Binary Tree. The first solution that we will look at applies to any kind of binary tree because it does not rely on any special properties of the tree. Our second solution will take into account the sorted nature of the binary search tree and will thus, improve upon the overall time complexity of the previous solution. The inorder successor of a particular node is simply the node that comes after this node during the inorder traversal of the tree. There are a few scenarios that we must consider for the inorder successor of a node to understand our first algorithm properly.





Simplest of all cases, the in
successor of the node 2 is its
child, 3 since there are no
nodes after that and that is
node which would come after
the inorder traversal of this

Figure 1. Few examples of inorder successors.



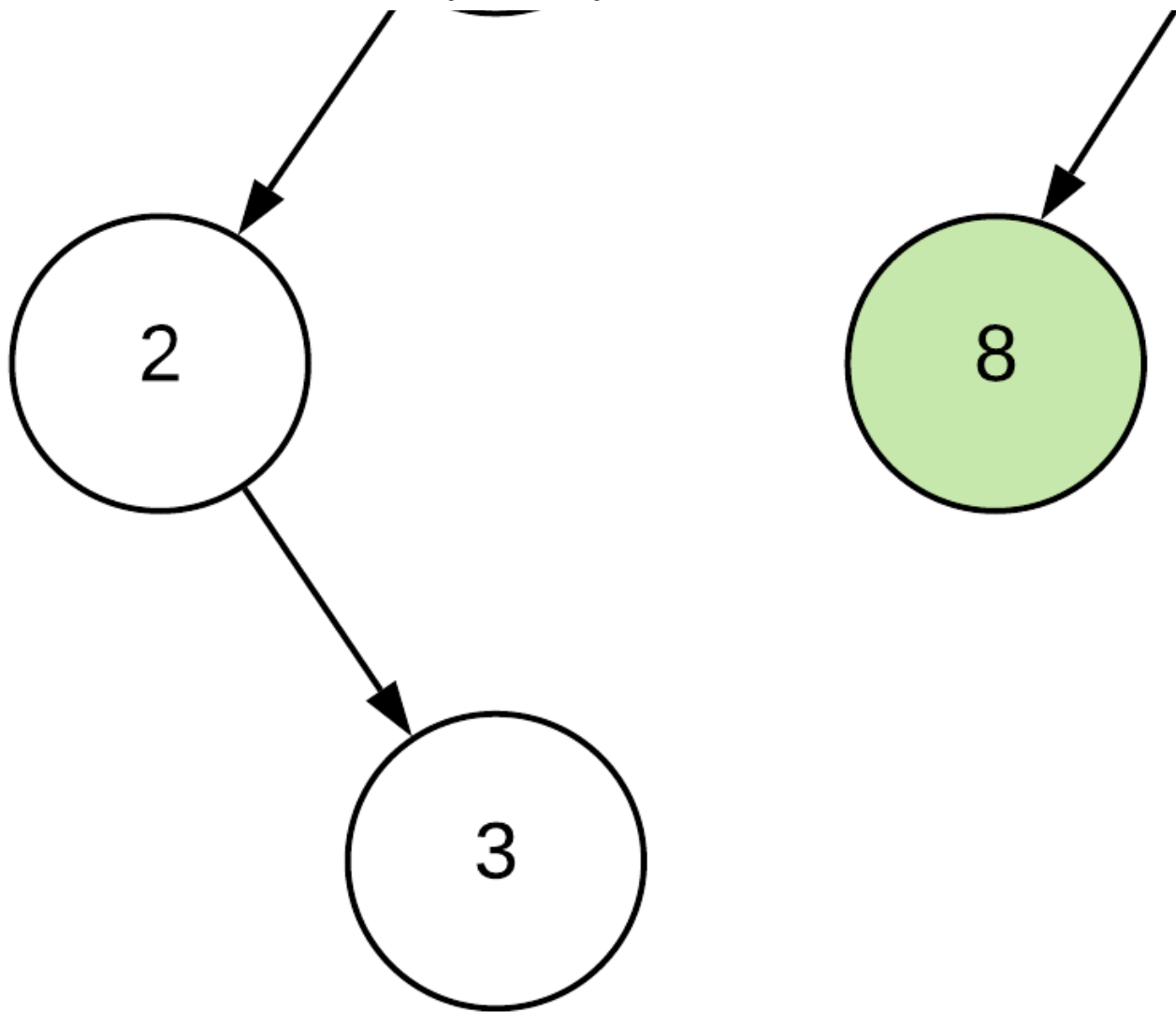


Figure 2. Another unique example of an inorder successor.

Approach 1: Without using BST properties

Intuition

As mentioned in the overview section of this article, we will first discuss the approach that applies to any binary tree and is not specifically for a binary search tree. This is not the most efficient approach out there considering it doesn't incorporate the search properties associated with the structure of a binary search tree. However, for the sake of completeness, we are including this approach in the official solution since the interviewer may ask you to find the inorder successor for a binary tree :)

We hinted briefly at the different cases for the inorder successor and we will look at these cases more concretely in this solution. The algorithm is based on handling these cases one by one. There are just two cases that we need to account for in this approach.

When the node has a right child

The inorder successor in this case is the leftmost node in the tree rooted at the right child. Let's look at a couple of examples to depict this point.

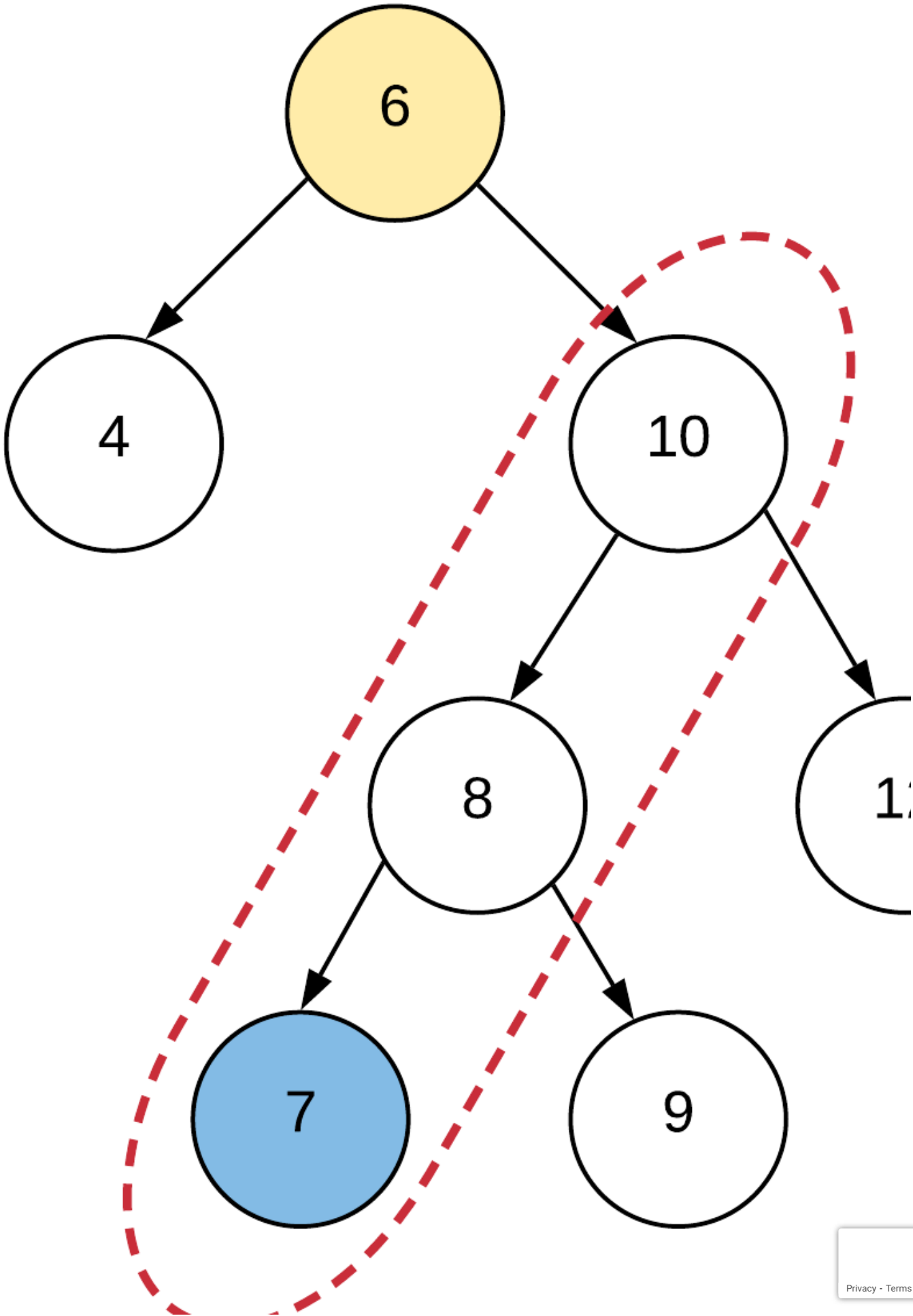




Figure 3. Case 1 when the given node has a right child.

Let's look at yet another example where there is a right child who doesn't have a left child. In this case, the right child itself will be the inorder successor of the designated node.

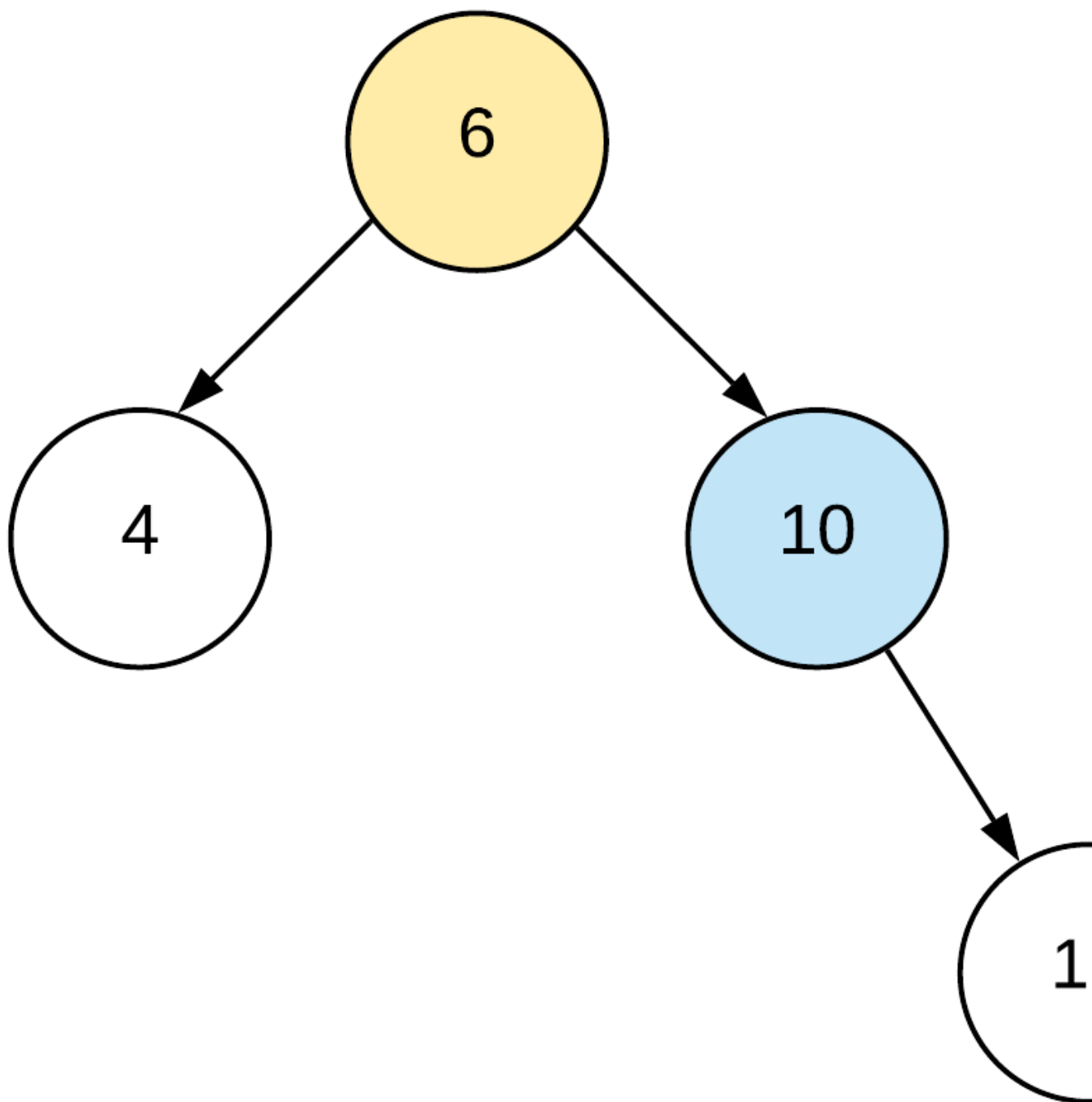


Figure 4. Another example of when the node has a right child.

When the node doesn't have a right child

This is trickier to handle than the first case. In this case, one of the ancestors acts as the inorder successor. That ancestor can be the immediate parent, or, it can be one of the ancestors further up the tree.

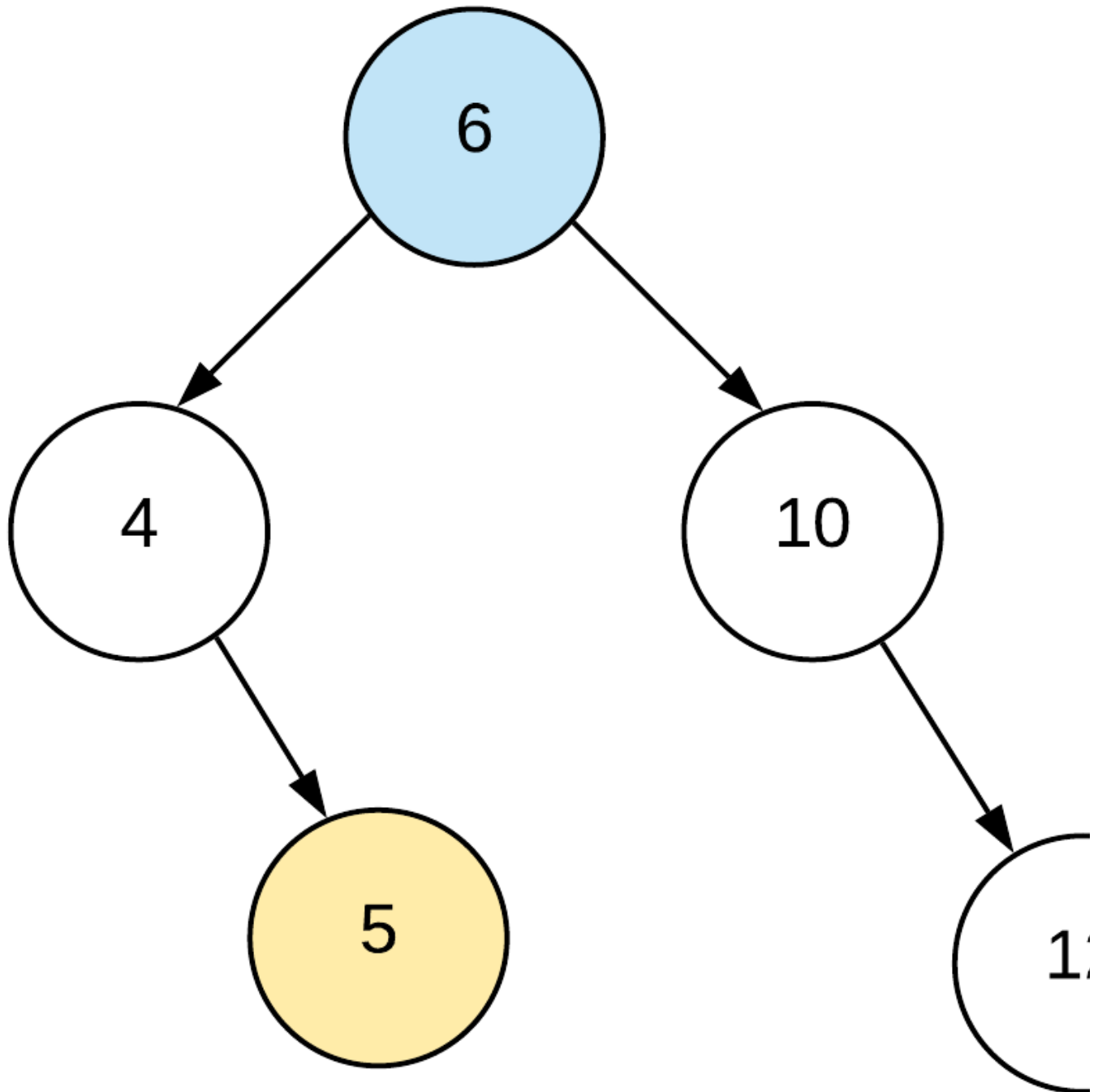


Figure 5. When the node does not have the right child.

In this case, we need to perform the inorder traversal on the tree and keep track of a previous node which is the predecessor to the current node we are processing. If at any point the predecessor `previous` is equal to the node given to us then the current node will be its inorder successor. Why? Because we are performing the inorder traversal on the tree to find the successor node via simulation.

Algorithm

1. We define two class variables for this algorithm: `previous` and `inorderSuccessorNode`. The `previous` variable will be used when handling the second case as previously explained and the `inorderSuccessorNode` will ultimately contain the node to be returned.

2. Inside the function `inorderSuccessor`, we first check which of the two cases we need to handle. For that, we simply check for the presence of a right child.

- *The right child exists*

In this case, we assign the right child to a node called `leftmost` and we iterate until we reach a node (`leftmost`) which doesn't have a left child. We iteratively assign `leftmost = leftmost.left` and that's how we will get the leftmost node in the subtree.

- *The right child does not exist*

1. As mentioned before, this case is trickier to handle. For this, we define another function called `inorderCase2` and we will pass it a node and the node `p`.
2. We perform simple inorder traversal and hence, we first recurse on the left child of the node.
3. Then, when the recursion returns, we check if the class variable `previous` is equal to the node `p`. If that is the case, then it means `p` is the inorder predecessor of node or in other words, the node is the inorder successor of the node `p` and we return from that point onwards. We assign `inorderSuccessorNode` to node and return from this function.

3. Finally, we return the `inorderSuccessorNode` as our result.

Implementation

Java

Python3

Copy

```

1 class Solution {
2
3     private TreeNode previous;
4     private TreeNode inorderSuccessorNode;
5
6     public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {
7
8         // Case 1: We simply need to find the leftmost node in the subtree rooted at p.right.
9         if (p.right != null) {
10
11             TreeNode leftmost = p.right;
12
13             while (leftmost.left != null) {
14                 leftmost = leftmost.left;
15             }
16
17             this.inorderSuccessorNode = leftmost;
18         } else {
19
20             // Case 2: We need to perform the standard inorder traversal and keep track of the previous node.
21             this.inorderCase2(root, p);
22         }
23
24         return this.inorderSuccessorNode;
25     }
26
27     private void inorderCase2(TreeNode root, TreeNode p) {

```

Complexity Analysis

- Time Complexity: $O(N)$ where N is the number of nodes in the tree.
 - For case 1, we might have a scenario where the root node has a right subtree that is left-skewed. Something like the following.

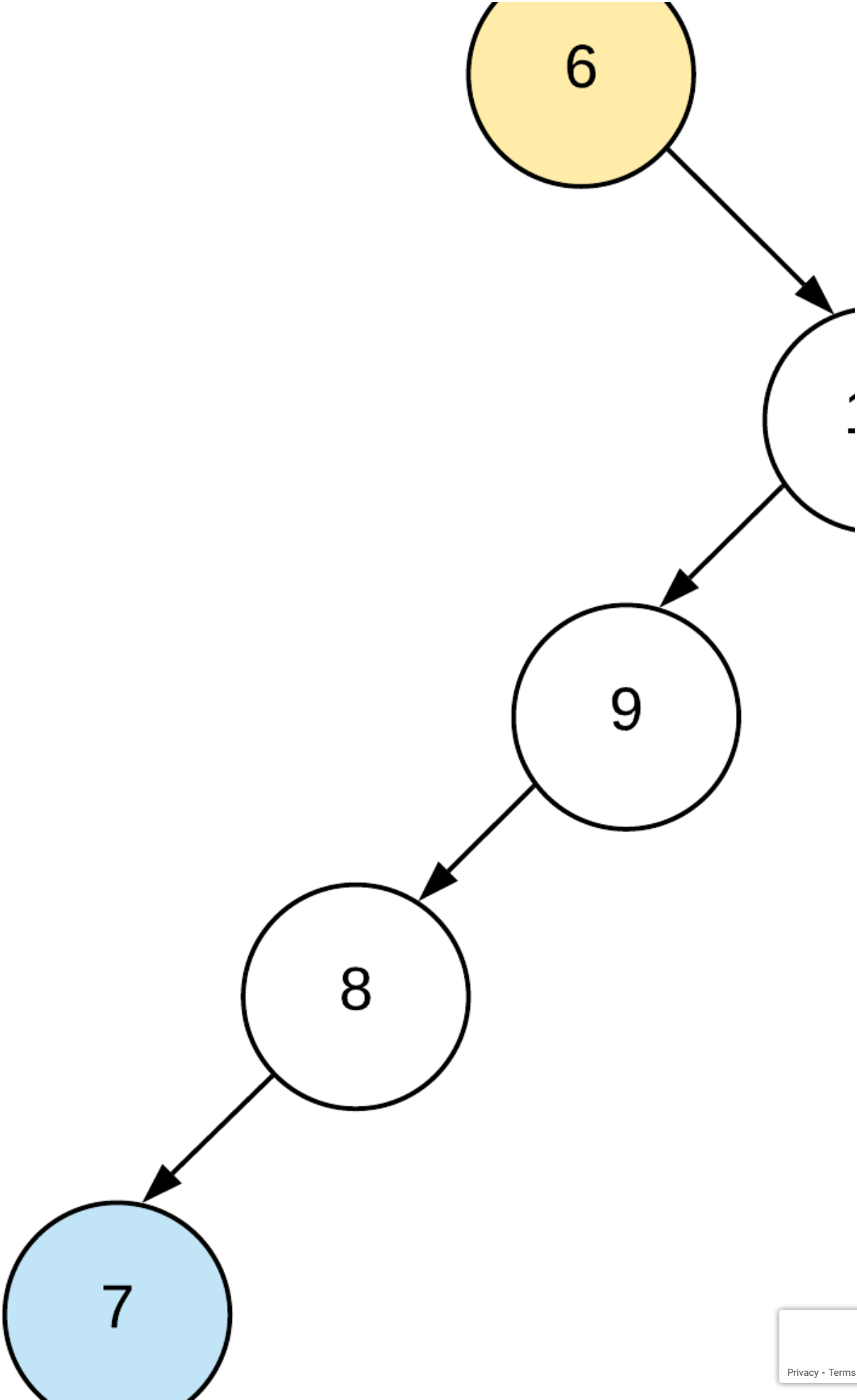
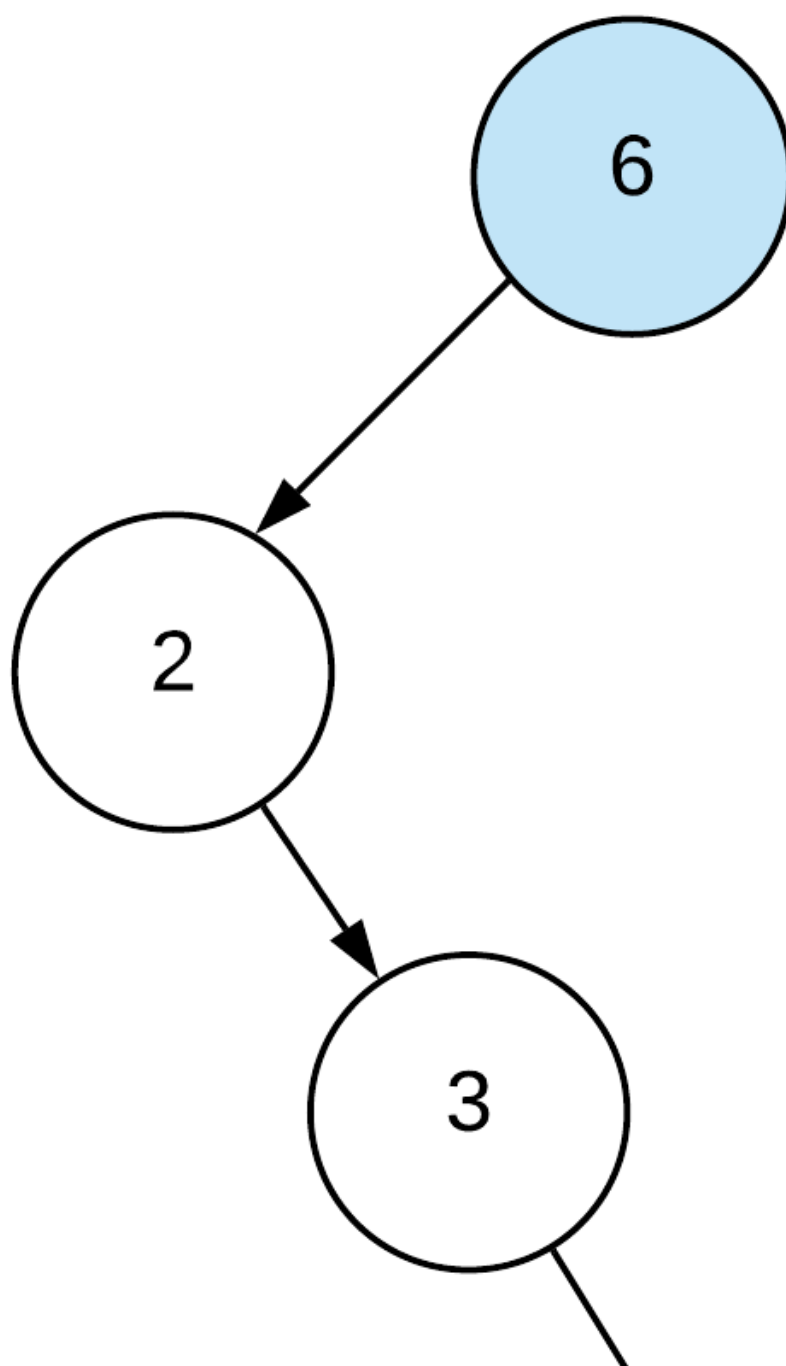


Figure 6. A skewed tree for worst-case time complexity.

In this case, we have to process all of the nodes to find the leftmost node and hence, the overall time complexity is $O(N)$.

- For case 2, we might have to process the entire tree before finding the inorder successor. Let's look at an example tree to understand when that might happen.



The
tree
find
pro
wo

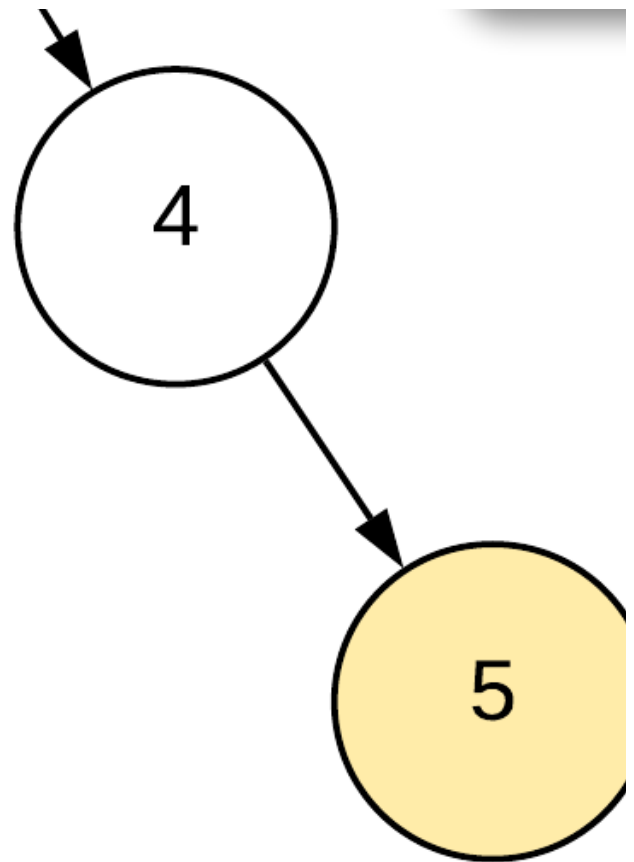


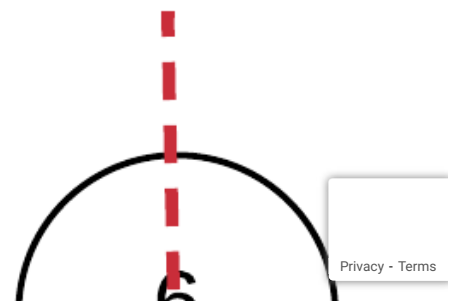
Figure 7. A skewed tree for worst-case time complexity.

- Space Complexity: Space Complexity: $O(N)$ for the second case since we might have a skewed tree leading to a recursion stack containing all N nodes. For the first case, we don't have any additional space complexity since we simply use a while loop to find the successor.

Approach 2: Using BST properties

Intuition

In the previous approach, we did not use any of the binary-search tree properties. However, the optimal solution for this problem comes from utilizing those properties and that's what we will explore in this solution. Specifically, we'll make use of the standard BST property where the left descendants have smaller values than the current node and right descendants have larger values than the current node. We don't need to handle any specific cases here and we can start with the root node directly and reach our inorder successor. Let's see the choices we have when comparing the value of the given node p to the current node in the tree.



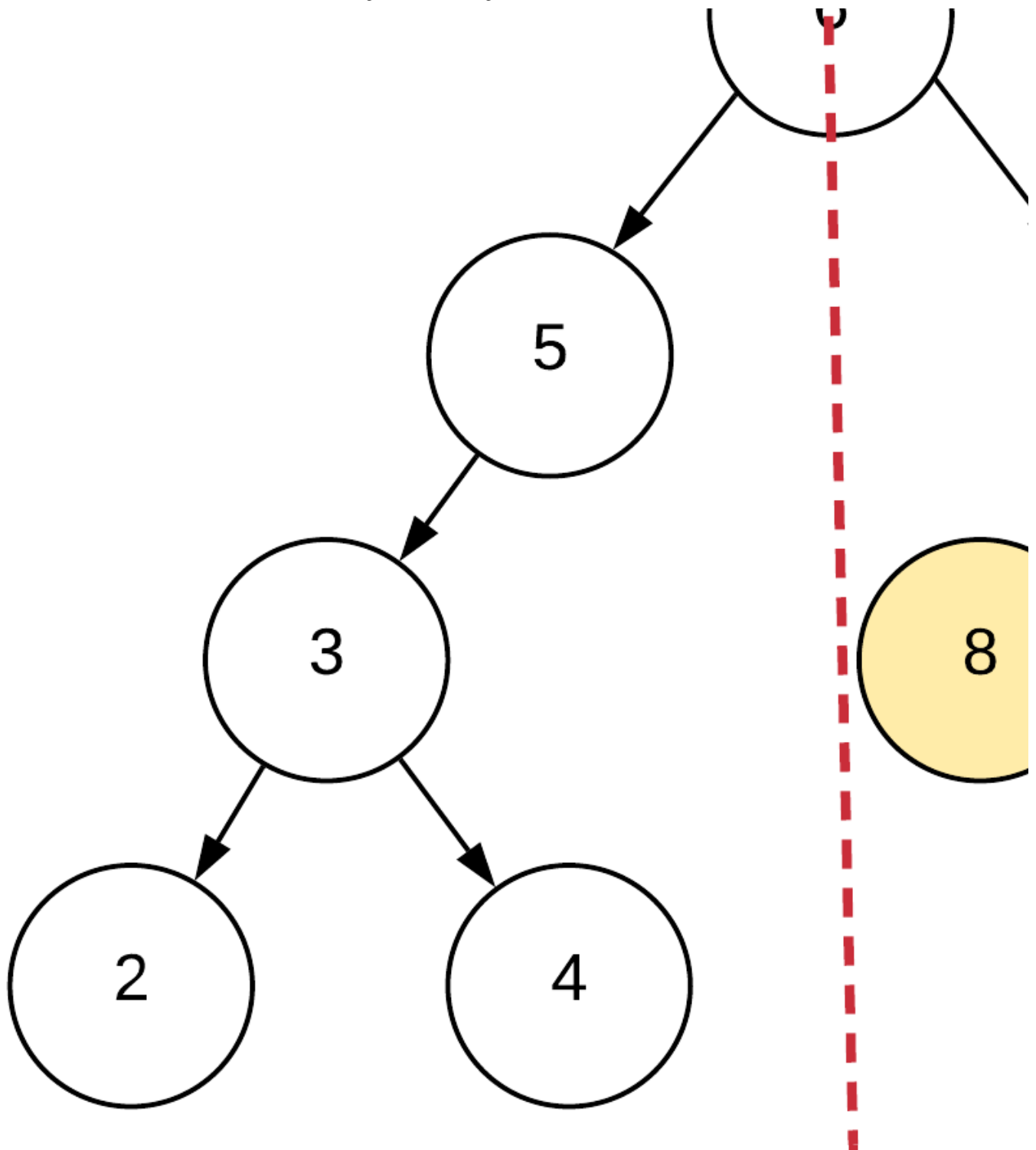


Figure 8. Skipping half of the binary search tree.

By comparing the values of the node p and the current node in the tree during our traversal, we can discard half of the remaining nodes at each step, and thus, for a balanced binary search tree we can search for our inorder successor in logarithmic time rather than linear time. That's a huge improvement over the previous solution.

Algorithm

1. We start our traversal with the root node and continue the traversal until our current node reaches a null value i.e. there are no more nodes left to process.

2. At each step we compare the value of node p with that of node.

1. If $p.val \geq node.val$ that implies we can safely discard the left subtree since all the nodes there including the current node have values less than p .

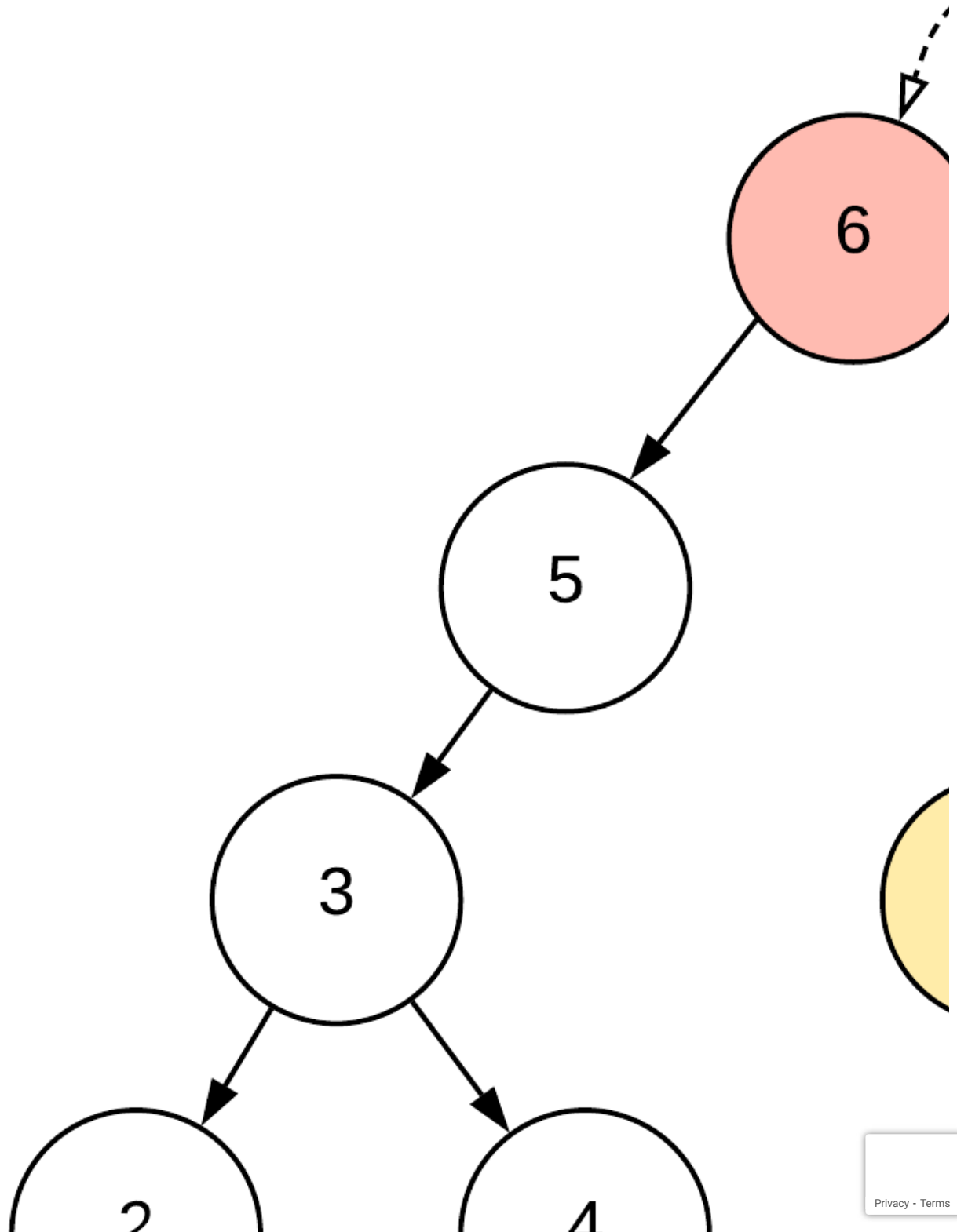




Figure 9. Skipping the left subtree.

2. However, if $p.val < node.val$, that implies that the successor must lie in the left subtree *and* that the current node is a ***potential candidate for inorder successor***. Thus, we update our local variable for keeping track of the successor, `successor`, to `node`.

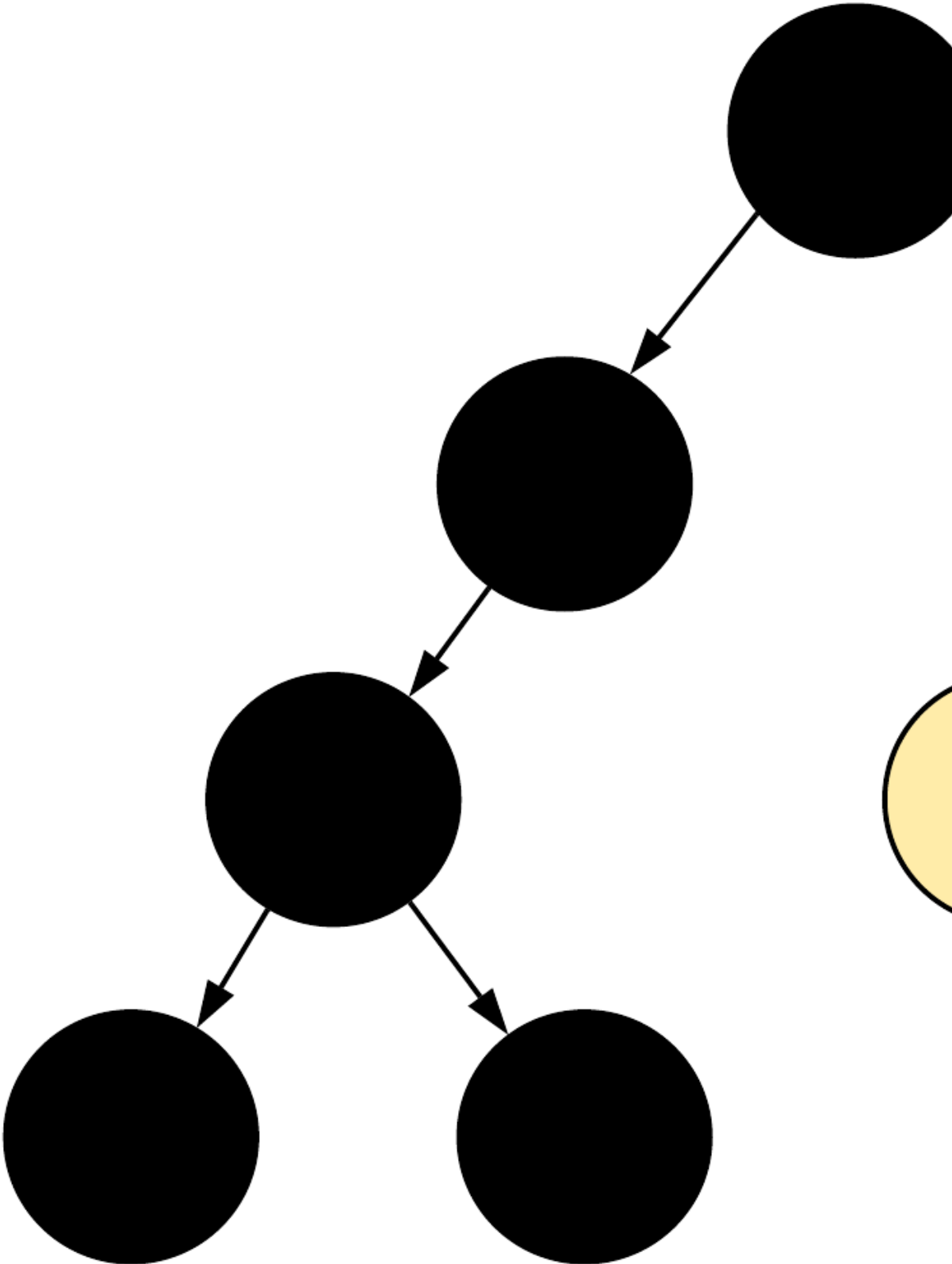


Figure 10. Skipping the right subtree and recording a potential candidate for the successor.

3. Return successor.

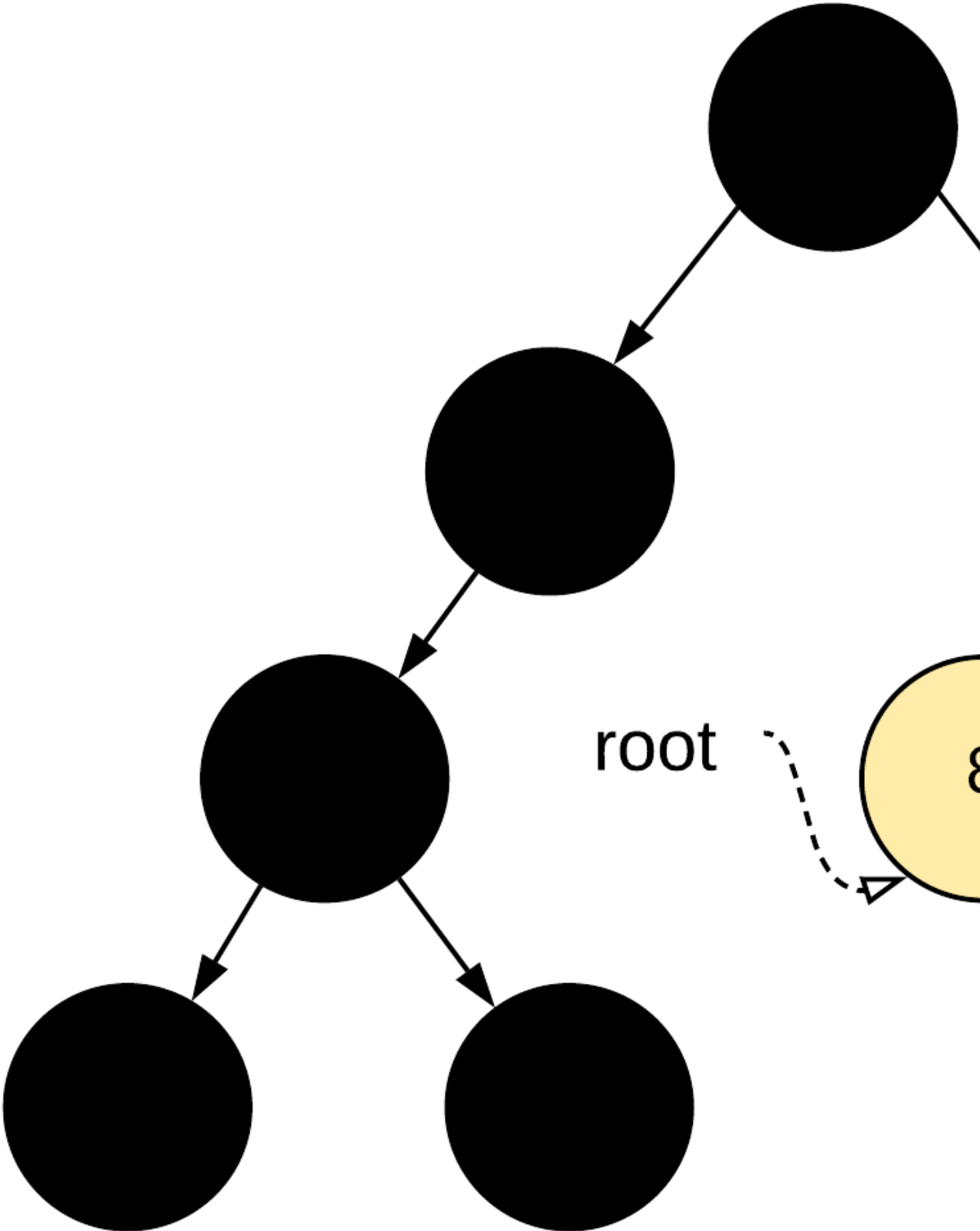


Figure 11. Returning the candidate.

We don't handle duplicate node values in the algorithm below. That is left as an exercise for the reader to solve :) It's a slight variation but an important one to understand for follow-up questions in an interview.

Implementation

```

Java Python3
1 class Solution {
2
3     public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {
4
5         TreeNode successor = null;
6
7         while (root != null) {
8
9             if (p.val >= root.val) {
10                 root = root.right;
11             } else {
12                 successor = root;
13                 root = root.left;
14             }
15         }
16
17         return successor;
18     }
19 }

```

Copy

Complexity Analysis

- Time Complexity: $O(N)$ since we might end up encountering a skewed tree and in that case, we will just be discarding one node at a time. For a balanced binary-search tree, however, the time complexity will be $O(\log N)$ which is what we usually find in practice.
- Space Complexity: $O(1)$ since we don't use recursion or any other data structures for getting our successor.

[Report](#) [Article](#) [Issue](#)

Comments: 39



☒ Best ☐ Most Votes ☐ Newest to Oldest ☐ Oldest to Newest

Type comment here...
(Markdown is supported)

Preview

Post



[klutch](#) ★ 175

[Privacy](#) - [Terms](#)

August 16, 2021 6:17 AM

Read More

That 2nd solution is beautiful

^

169

v

Show 2 replies

Reply

Share

Report

[bwaarl](#) ★41

September 25, 2021 8:21 AM

Read More

How does the first solution not use BST properties? If we assume that the leftmost child of the right subtree is the successor, that's assuming BST principles. Cuz if it was any old binary tree, the successor could be anywhere, it could be the the rightmost child for all we know.

^

40

v

Show 10 replies

Reply

Share

Report

[battleslug](#) ★34

Last Edit: June 28, 2021 5:37 AM

Read More

Using stack, doesn't matter if the tree is a valid BST or not

```
class Solution {
    public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {
        Stack<TreeNode> s = new Stack<>();
```

```

TreeNode node = root;
boolean found = false;
while (node != null || !s.isEmpty()) {
    while (node != null) {
        s.push(node);
        node = node.left;
    }

```

```

    node = s.pop();
    if (found) {
        return node;
    }
    if (node.val == p.val) {
        found = true;
    }
    node = node.right;
}

```

[Show 2 replies](#)
[Reply](#)
[Share](#)
[Report](#)



[brucewen05](#) ★17

March 28, 2021 1:32 AM

[Read More](#)

Can someone help me understand how to tweak solution 2 to handle duplicates?

Does it depend on how we treat duplicate values? i.e. whether we choose to insert a node with equal value to its parent on the left or right?

It seems to me solution 1 is a more general solution although it uses more space due to recursion.

[Show 8 replies](#)
[Reply](#)
[Share](#)
[Report](#)



[devanother](#) ★23

April 18, 2021 11:10 AM

[Read More](#)

I got both, the iterative and the recursive approaches and felt like my recursive approach was simpler than what they have put here. My iterative one was quite similar to their but still simpler imo.

```
class Solution {
    public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {
        if(root == null){
            return null;
        }

        TreeNode successor = inorderSuccessor(root.left, p);
        if(successor != null){
            return successor;
        }
        if(root.val > p.val){
            return root;
        }
        return inorderSuccessor(root.right, p);
    }
}
```

↩️ Reply }

📄 Share

⚠️ Report



[Goldspear](#) ★92

Last Edit: April 8, 2021 7:57 PM

Read More

Combine the two solutions to allow earlier stopping:

1. If p has a right leaf, we only need to traverse from p to leaf (one step right, all step left)
2. Else, we only need to traverse from root to p

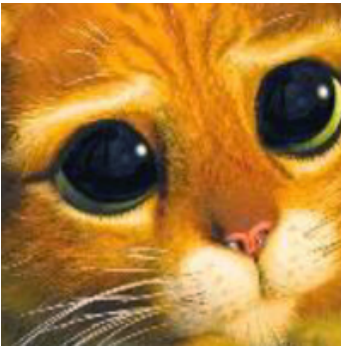
```
def inorderSuccessor_02(root: 'TreeNode', p: 'TreeNode') -> 'TreeNode':
    if p.right:
        curr = p.right
        while curr.left:
            curr = curr.left
        return curr
    else:
        successor, curr = None, root
        while curr != p:
            if curr.val < p.val:
                curr = curr.right
            else:
                successor, curr = curr, curr.left
        return successor
```

🗨️ Show 3 replies

↩️ Reply

📄 Share

⚠️ Report



[jjzzzmd](#) ★12

Last Edit: December 9, 2021 7:14 PM

Read More

For Sol 1, I don't understand why go through all the trouble to analyze case by case where we can do an iterative in order traversal, same time and space complexity and much easier to understand (thus easier to come out in an interview).

```
class Solution:
    def inorderSuccessor(self, root: 'TreeNode', p: 'TreeNode') -> 'Optional[TreeNode]':
        stack = []
        take = False
        while True:
            while root:
                stack.append(root)
                root = root.left
            if not stack:
                return None
            root = stack.pop()
            if root==p:
                take = True
            elif take:
                return root
            root = root.right
```

↩ Reply

📄 Share

⚠ Report

return None



[unknown1234](#) ★11

Last Edit: May 25, 2021 12:21 PM

Read More

(Not using BST property) just take the next of p during the inorder traversal:

```
def inorderSuccessor(self, root: 'TreeNode', p: 'TreeNode') -> 'TreeNode':
    curr=root
    stack=[]
    turn=False
    while curr or stack:
        while curr:
            stack.append(curr)
            curr=curr.left
        # curr=None
        # stack top is the left most
```



```

    ^       node=stack.pop()
11         if node.right:
    v             curr=node.right

```

Show 2 replies

↩ Reply

🔖 Share

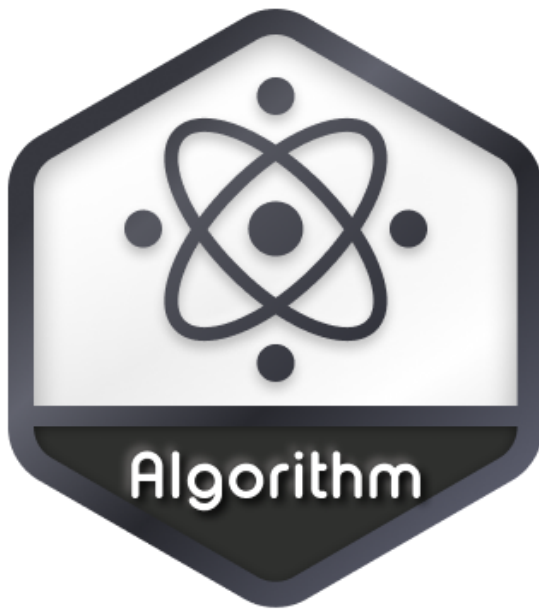
⚠ Report

```

    turn:
    return node

    if node==p:
    turn=True

```



★3

[luismendoza](#)

March 16, 2022 11:40 AM

Read More

We could keep track of the last node visited in a global variable, which would allow us to tell if the current node is the successor. Making recursive code very intuitive.

```

public class Solution {
    private TreeNode lastSeen;
    private TreeNode found;

    public TreeNode InorderSuccessor(TreeNode root, TreeNode p) {
        Recurse(root, p);
        return found;
    }

    private void Recurse(TreeNode root, TreeNode p) {
        if(root is null) {
            return;
        }
        Recurse(root.left, p);
        if(lastSeen == p) {
            found = root;
        }
    }
}

```



dileep_reddy_9★1

Last Edit: February 19, 2022 10:24 AM

Read More

Optimization on approach 1, insert the following lines at Line#32 in the Java solution.

```
// don't traverse the rest of the tree if the successor node is already found
if(this.inorderSuccessorNode != null) return;
// avoid traversing the left subtree of p unnecessarily. Right subtree of p is null anyway at this point.
if(node == p) {
    this.previous = p;
    return;
}
```

- ^
- 1
- ▼
- ↩ Reply
- 📄 Share
- ⚠ Report

-
- 1
- 2
- 3
- 4
-

Time Submitted **Status** **Runtime** **Memory** **Language**
10/05/2022 23:32 [Accepted](#) 72 ms 22.9 MB cpp

☰ Problems

✂ Pick One

< Prev

285/2430

Next >

i

C++

✓

Autocomplete

i

{ }

↺

🔍

⌕

```
xxxxxxxxxx
```

```
53
```

```
};
```

```
1
```

```
/**
```

```
2
```

```
 * Definition for a binary tree node.
```

```
3
4
5 * struct TreeNode {
6
7     int val;
8
9     TreeNode *left;
10
11    TreeNode *right;
12
13    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
14
15 };
16
17
18
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
```

```
//    }  
24  
  
25  
//    void traverse(TreeNode* root, TreeNode* p) {  
26  
//        if (!root) return;  
27  
//        traverse(root->left, p);  
28  
//        if (cache == p && !res) {  
29  
//            cout<<root->val<<endl;  
30  
//            res = root; return; }  
31  
//        cache = root;  
32  
//        traverse(root->right, p);  
33  
//    }  
34  
// };  
35  
  
36  
  
37  
class Solution {  
38  
    TreeNode* cache = NULL;  
39  
public:  
40  
    TreeNode* inorderSuccessor(TreeNode* root, TreeNode* p) {  
41  
        if (p->right) {
```

Your previous code was restored from your local storage. [Reset to default](#)

✕

Console ▾

Contribute

i

▶ Run Code ^ Submit

❏

///

Type here... (Markdown is enabled)

✕

Saved

Privacy - Terms

All Problems

1



✓ #1 Two Sum

Easy

✓ #2 Add Two Numbers

Medium

✓ #3 Longest Substring Without Repeating Characters

Medium

#4 Median of Two Sorted Arrays

Hard

✓ #5 Longest Palindromic Substring

Medium

✓ #6 Zigzag Conversion

Medium

✓ #7 Reverse Integer

Medium

✓ #8 String to Integer (atoi)

Medium

✓ #9 Palindrome Number

Easy

#10 Regular Expression Matching

Hard

✓ #11 Container With Most Water

Medium

✓ #12 Integer to Roman

Medium

✓ #13 Roman to Integer

Easy

✓ #14 Longest Common Prefix

Easy

✓ #15 3Sum

Medium

#16 3Sum Closest

Medium

✓ #17 Letter Combinations of a Phone Number

Medium

✓ #18 4Sum

Medium

✓ #19 Remove Nth Node From End of List

Medium

✓ #20 Valid Parentheses

Easy

✓ #21 Merge Two Sorted Lists

Easy

✓ #22 Generate Parentheses

Medium

✓ #23 Merge k Sorted Lists

Hard

#24 Swap Nodes in Pairs

Medium

✓ #25 Reverse Nodes in k-Group

Hard

✓ #26 Remove Duplicates from Sorted Array

Easy

✓ #27 Remove Element

Easy

✓ #28 Find the Index of the First Occurrence in a String

Medium

✓ #29 Divide Two Integers

Medium

#30 Substring with Concatenation of All Words

Hard

✓ #31 Next Permutation

Medium

#32 Longest Valid Parentheses

Hard

✓ #33 Search in Rotated Sorted Array

Medium

#34 Find First and Last Position of Element in Sorted Array

Medium

✓ #35 Search Insert Position

Easy

#36 Valid Sudoku

Medium

#37 Sudoku Solver

Hard

#38 Count and Say

Medium

✓ #39 Combination Sum

Medium

✓ #40 Combination Sum II

Medium

#41 First Missing Positive

Hard

#42 Trapping Rain Water

Hard

✓ #43 Multiply Strings

Medium

#44 Wildcard Matching

Hard

✓ #45 Jump Game II

Medium

✓ #46 Permutations

Medium

#47 Permutations II

Medium

✓ #48 Rotate Image

Medium

✓ #49 Group Anagrams

Medium

#50 Pow(x, n)

Medium

