UTS DATA STRUCTURE BOOTCAMP

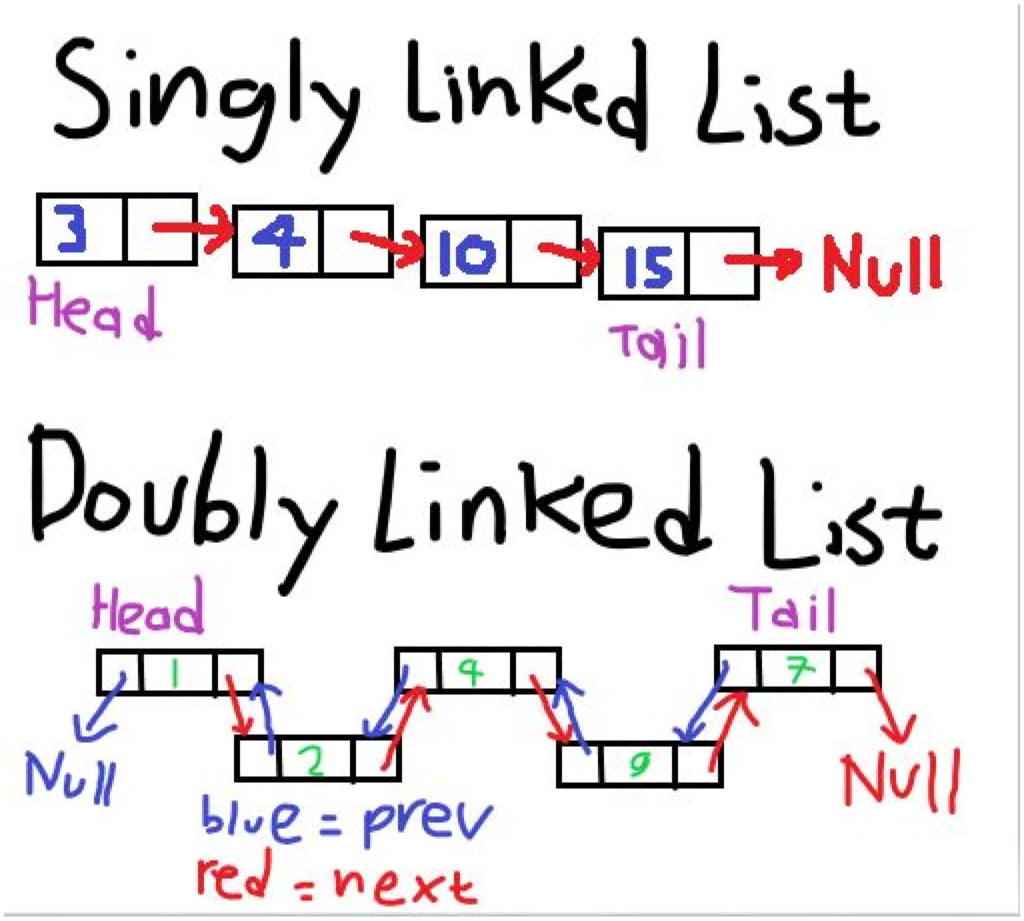
Answer by: **Complexx (Complexx#3616)**

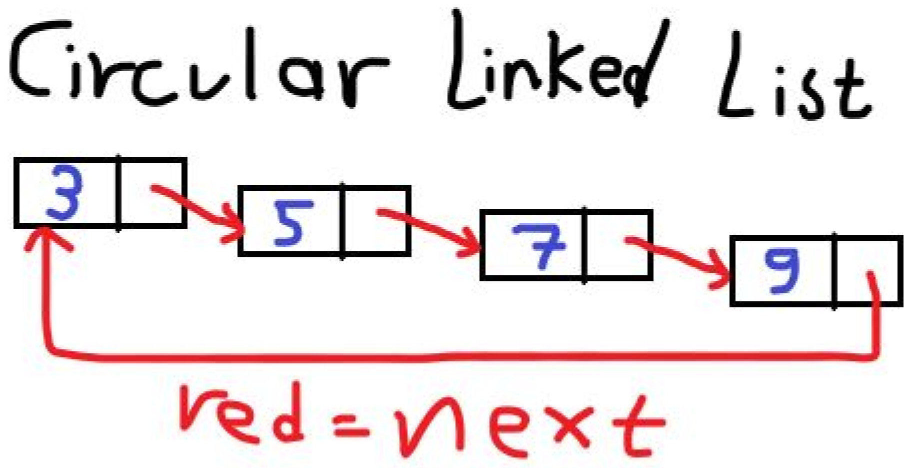
Group: **Chuunibyou Club**

"*Overpowered dan senang ngehalu*"

A. Linked List

1. Explain single, double, and circular linked list in a graphical view!





2. **What are the main differences between Linked List and Array?**

   Array:
   - Elements are stored in consecutive memory location, referenced by an index.
   - Index started from 0.
   - Fixed size
   - O(N) time to insert/delete and search
   - O(1) time to access

   Linked List:
   - Elements can be stored anywhere in the memory.
   - Does not use index (unlike array)
   - Flexible size
   - O(N) time to access and search
   - O(1) time to insert/delete

3. **Explain Floyd's algorithm and implementation including its pseudocode!**

   In linked list, there's Floyd's cycle detection algorithm that detects a cycle or a loop in a linked list (note that if you search for Floyd's algorithm, the search engine will redirect you to Floyd-Warshall algorithm to find the shortest path). Floyd's algorithm uses 2 pointers, a fast pointer, which moves 2 nodes at a time; and a slow pointer, which moves 2 nodes at a time. There will be 2 cases:

   a. <u>No loop present</u>

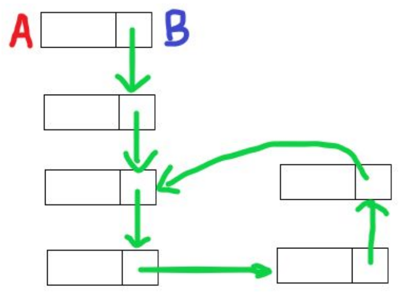
      If there's no loop, then the fast pointer will eventually reach the end of the linked list sooner than the slow pointer.

   b. <u>Loop present</u>

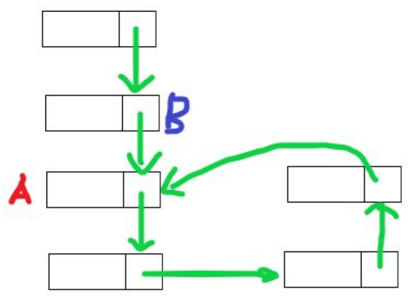
      If there's a loop, then the fast pointer will eventually meet up with the slow pointer at some point.

   Implementation:

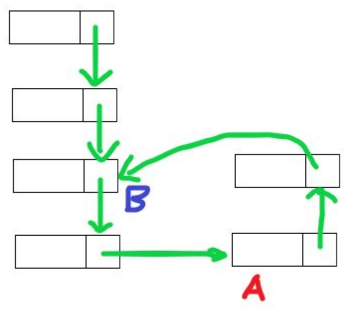
   1. Set 2 pointers (let's say pointer A is the fast one while pointer B is the slow one)
   2. Let pointer A and B traverse the whole linked list one by one
   3. Case:
      a. A reaches NULL (or end of Linked List): output "No cycle detected"
      b. A meet up with B: output "Cycle detected"
      c. Else: keep traversing



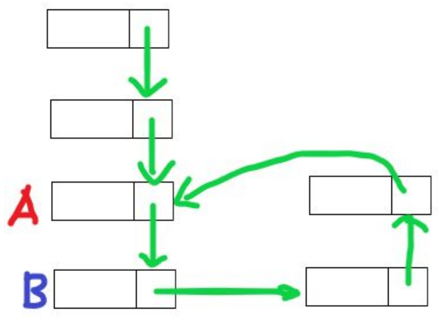
→



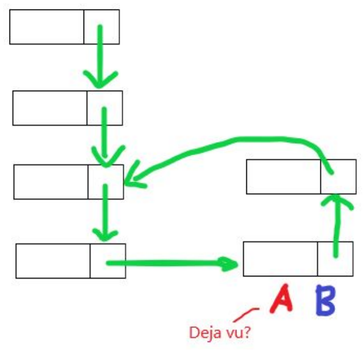
→



→



→



→ A meet up with B → **Loop Detected**

Pseudocode (function to detect cycle):

```
START
    cycledetection(Node *target){
        If ( target == NULL){
            // Empty Linked List, impossible to create a cycle
            print “No cycle detected”
            return
        }else if ( target→next == NULL ){
            // Only 1 node in Linked List, impossible to create a cycle
            print “No cycle detected”
            return
        }
        Node *A = target
        Node *B = target
        bool flag1 = false
                        //case no loop present
        bool flag2 = false
                        //case loop present
        while(!flag1 && !flag2){
            A = A→next→next
            B = B→next
            If (A ==  B){
                flag2 = true
            }else if ( A == NULL || A→ next == NULL){
                flag1 = true
            }
        }
        If ( flag1 ) print “No cycle detected”
        Else print “Cycle detected”
        return
    }
END
```

B. Stack and Queue

1. **What are the main differences between Stack and Queue?**
   Stack is where we stack elements on top of other elements just like stacking plates on top of other plates and we can only pop out the top most elements. Queue is where the elements are in the queue, just like waiting to get into the bus, if more passengers come, they queue from the back and the first one to get to the bus is the most front passenger. Stack applies First In Last Out (FILO) while Queue applies First In First Out (FIFO).

2. **Explain prefix, infix, and postfix notation and its implementation using stack!**

<u>**Infix notation**</u> is the conventional notation in which many people recognize.

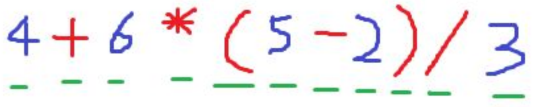
Example: 5 + 3 * 4 or 4 + 6 * (5 - 2) / 3

Infix notation is easy for humans to read, but harder for computers. Computers need to traverse the whole operation first to find the highest precedence, which could lead to O(N^2) time complexity.

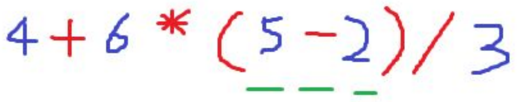
Implementation:

1. Traverse the whole operation, find the highest precedence
2. Calculate the operation with highest precedence operator:
    a. The highest: ()
    b. * or / (multiplication or division)
    c. The lowest: + or - (addition or subtraction)
3. Selection:
    a. If: found ‘(‘ and ‘)’ (the most outer ones) repeat step 1-2 for everything inside the ‘(‘ and ‘)’ , after all process inside ‘(‘ and ‘)’ is cleared, remove the brackets
    b. Else: otherwise calculate the one with highest precedence that appears first
4. Repeat step 1-3 until all operation is complete (no operators left)

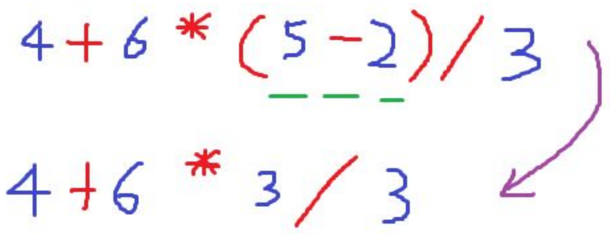
Traverse the whole operation



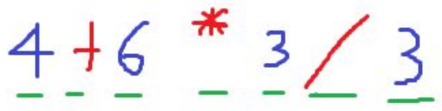
Highest precedence: ‘(‘ and ‘)’



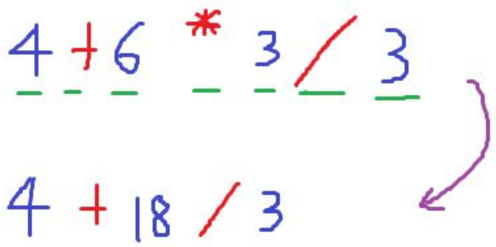
Highest precedence: - (subtraction)



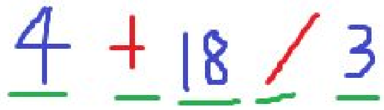
Traverse



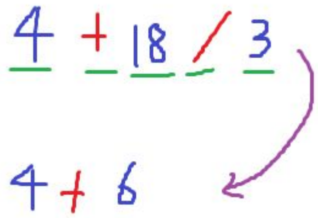
Highest precedence: * and / (but since * appear first, we do multiplication first)



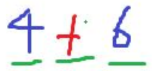
Traverse



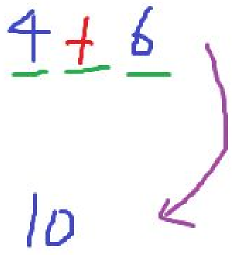
Highest precedence: / (division)



Traverse



Highest precedence: + (addition)

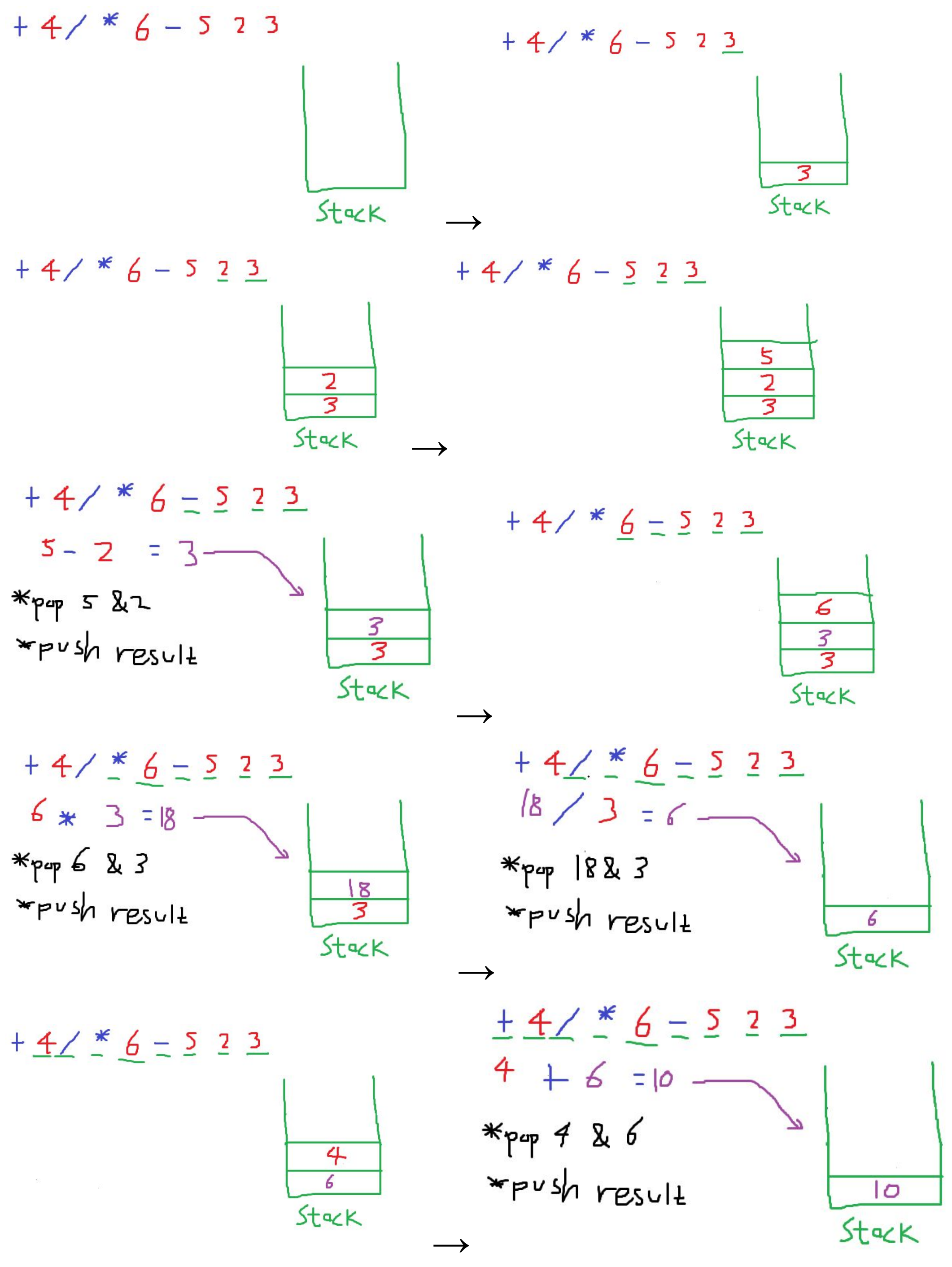


Traverse

No operators left, RESULT: 10

<u>Prefix notation</u> is where the operators came first, then followed by the 2 operands.
Example: + 5 * 3 4 or + 4 / * 6 - 5 2 3
Implementation:

1. Starts from the right side
2. If found operands, push to stack
3. If found operators, pop 2 operands, operate, then push the result to stack

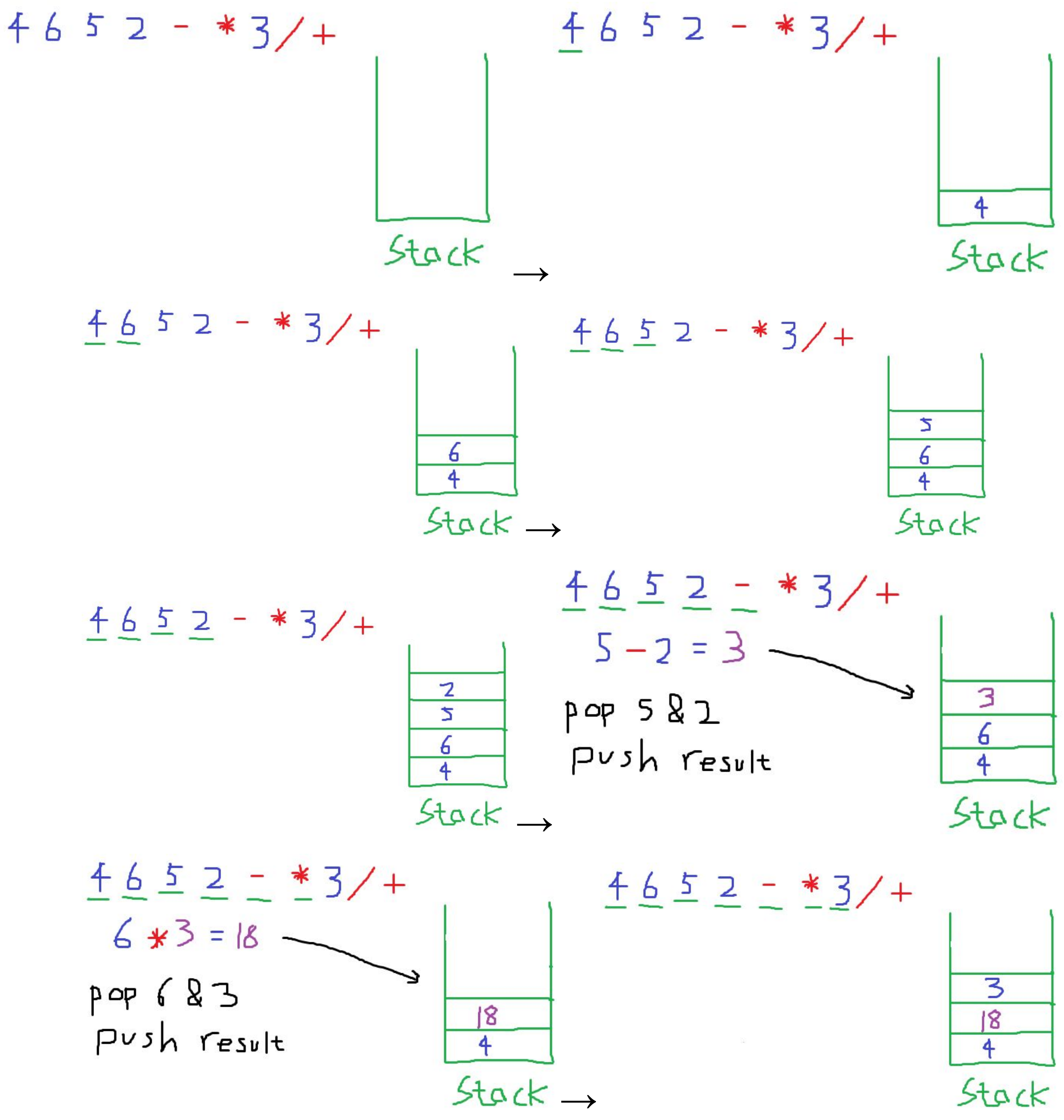


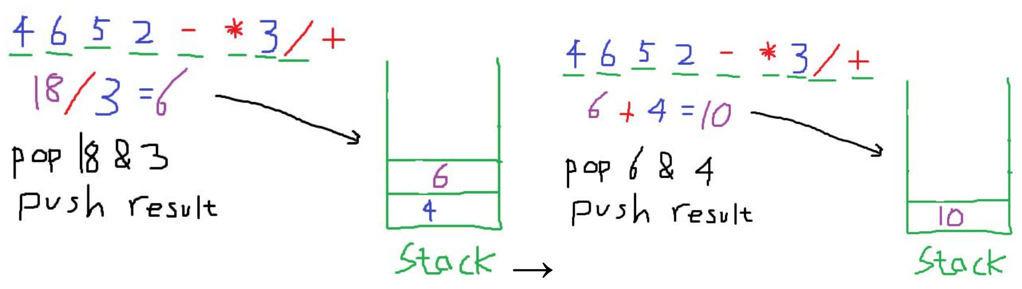
RESULT: 10

<u>Postfix notation</u> is where the 2 operands came first then followed by the operators.
Example: 5 3 4 * + or 4 6 5 2 - * 3 / +

implementation:

1. Starts from the left side
2. If found operands, push to stack
3. If found operators, pop 2 operands, operate, then push the result to stack





RESULT: 10

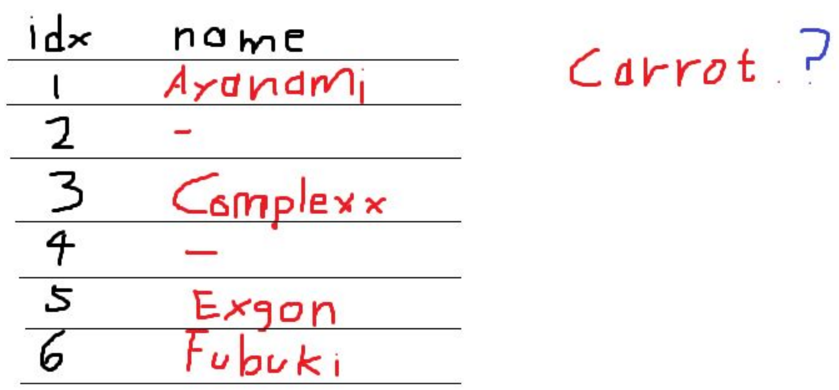
Both Prefix and Postfix notation is a bit harder for humans to read but easy for computers to read since all it need to do is to evaluate from left to right (postfix) or right to left (prefix)

C. Hashing and Hash Table

1. **Explain what is a hashtable, hash function, and collision!**
   Hashing is used to store and retrieve keys. In hashing, strings are stored in an array in a particular order where its index is the key. So to search for the string, all we need to do is to search for the key. The key can be obtained by using **Hash Function**.
   Hashtable is where we store the original strings (the array that stores the string).
   Hash Function is the function to transform strings into keys. The way we could transform (hash) it includes:
   - Mid-square
   - Division
   - Folding
   - Digit extraction
   - Rotating hash
   - First character of each strings
   - Etc.

   In some cases, multiple strings could have the same key and this is what we call a collision. **Collision** is a condition where multiple different strings have the same key while using the same hash function.

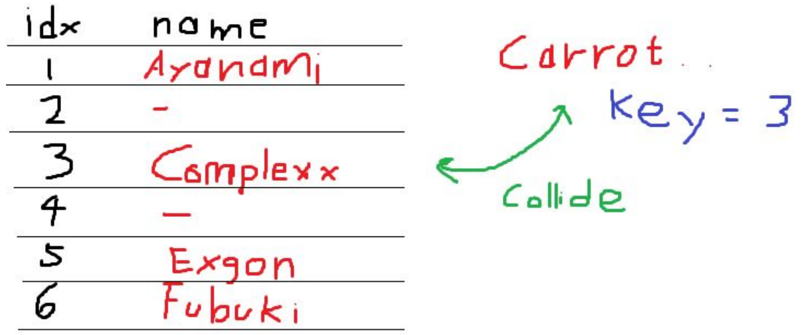
2. **Explain 2 methods for collision handling and simulate the process in a graphical view!**
   2 Methods for collision handling:
   a. Linear probing: we search for the next empty slot in the hashtable.
      i. Let's say we have a hashtable that can store 6 elements/strings inside. We already have "Ayanami", "Complexx", "Exgon", "Fubuki" inside the hashtable and we want to insert "Carrot". (idx = index, name = the string)



idx	name
1	Ayanami
2	-
3	Complexx
4	—
5	Exgon
6	Fubuki

ii.

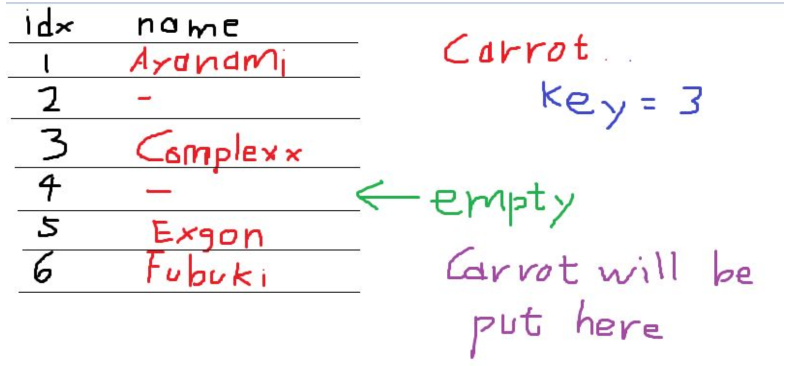
iii. The Hash Function makes the key by taking the first alphabet of the string and converting it into integers based on alphabetical order (in this case 'C' is the 3rd alphabet, so "Carrot" has 3 as the key, which collides with Complexx).



idx	name
1	Ayanami
2	-
3	Complexx
4	—
5	Exgon
6	Fubuki

iv.

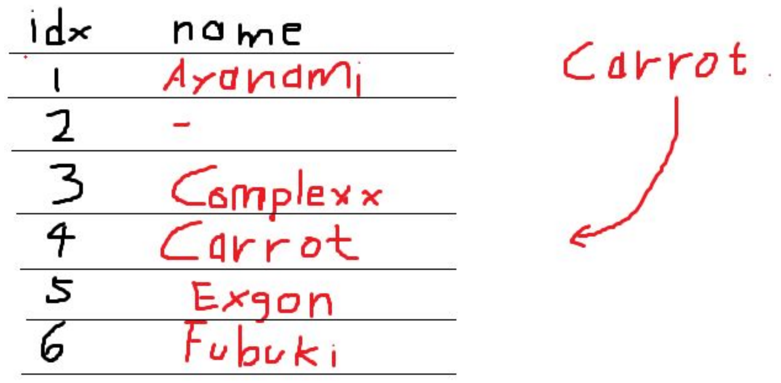
v. We find the next empty slot, which happens to be at index=4.



idx	name
1	Ayanami
2	-
3	Complexx
4	—
5	Exgon
6	Fubuki

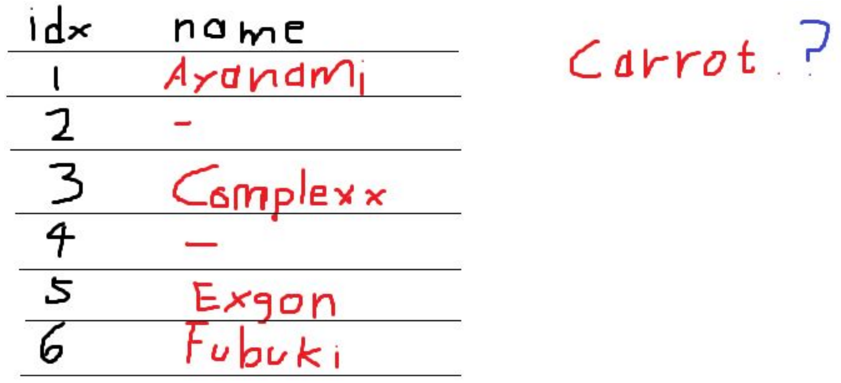
vi.

vii. We put "Carrot" in index 4.

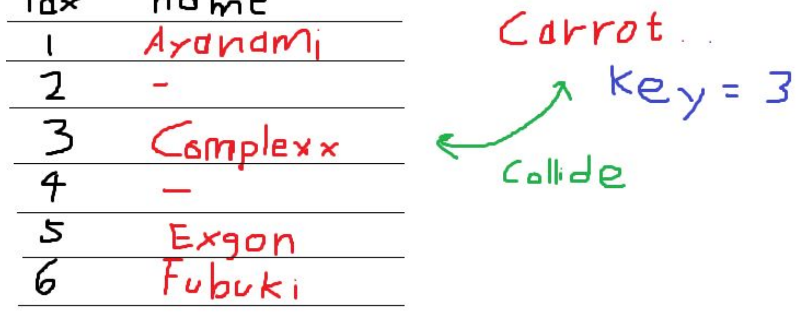


viii.

b. Chaining: we chain the strings in the array.
    i. Let's say we have a hashtable that can store 6 elements/strings inside. We already have "Ayanami", "Complexx", "Exgon", "Fubuki" inside the hashtable and we want to insert "Carrot". (idx = index, name = the string)

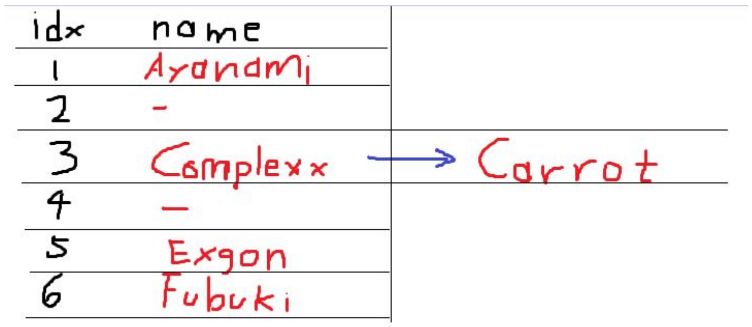


    ii.
    iii. The Hash Function makes the key by taking the first alphabet of the string and converting it into integers based on alphabetical order (in this case 'C' is the 3rd alphabet, so "Carrot" has 3 as the key, which collides with Complexx).



    iv.

v. We chain “Carrot” to “Complexx”



idx	name	
1	Ayanami	
2	-	
3	Complexx	→ Carrot
4	—	
5	Exgon	
6	Fubuki	

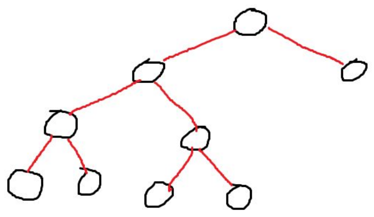
vi.

vii. Hence, there are 2 strings in index 3, linked using Linked List.

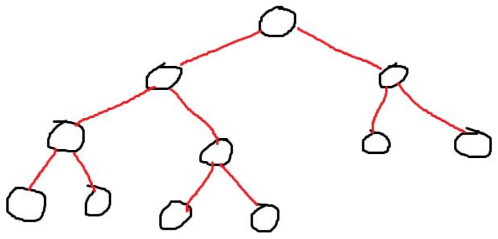
D. Binary Search Tree

1. Explain 5 types of Binary Tree and draw each of them! (BT = Binary Tree)

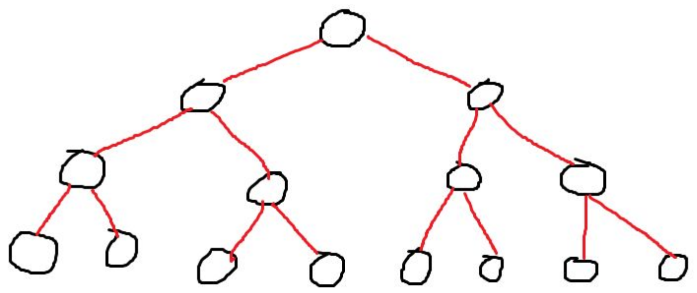
a. Full BT: BT that has either 0 or 2 children (all nodes are either full or empty).
Example:



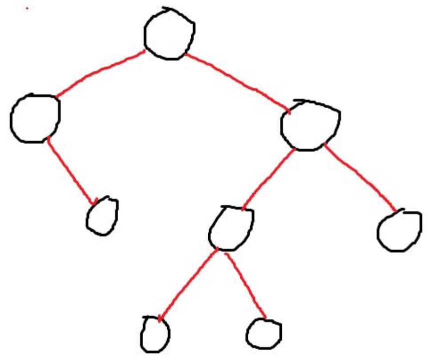
b. Complete BT: BT where all tree levels are filled entirely except for the last level.
Example:



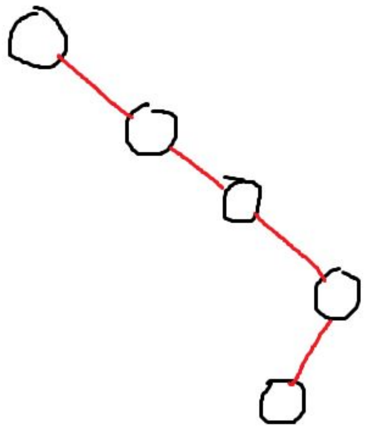
c. Perfect BT: BT where all nodes have strictly 2 children and all the leaves are the same. Example:



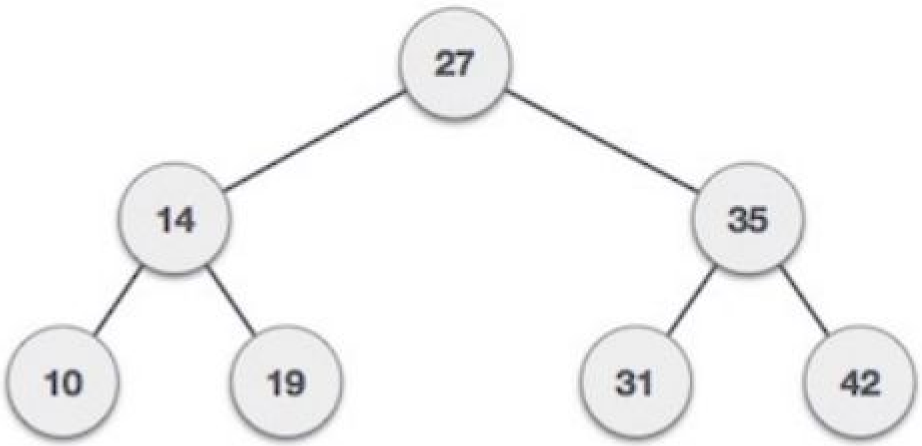
d. Balanced BT: BT with height O(log N) where N is the number of Nodes. One of its characteristics is the height difference between the left and the right tree is at most 1 (one). Example:



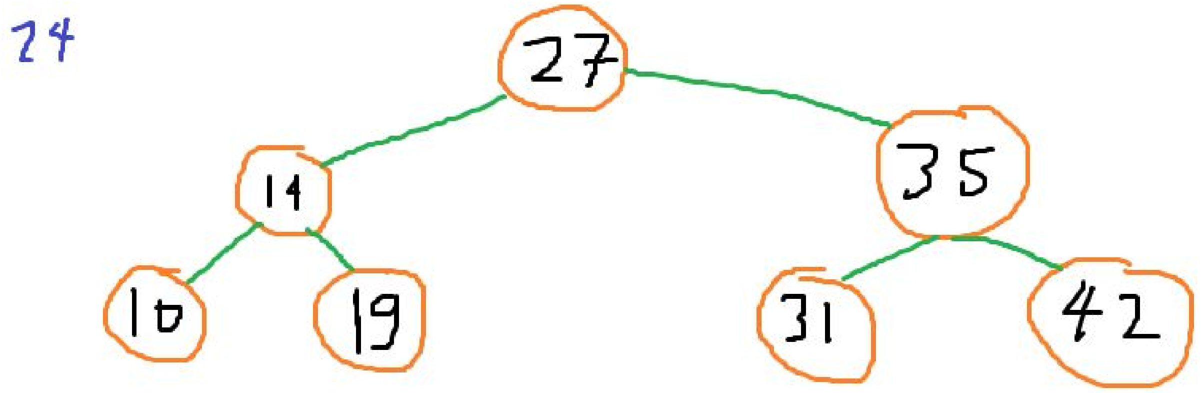
e. Degenerate BT: BT where every node has only one child, more or less similar to Linked List. Example:



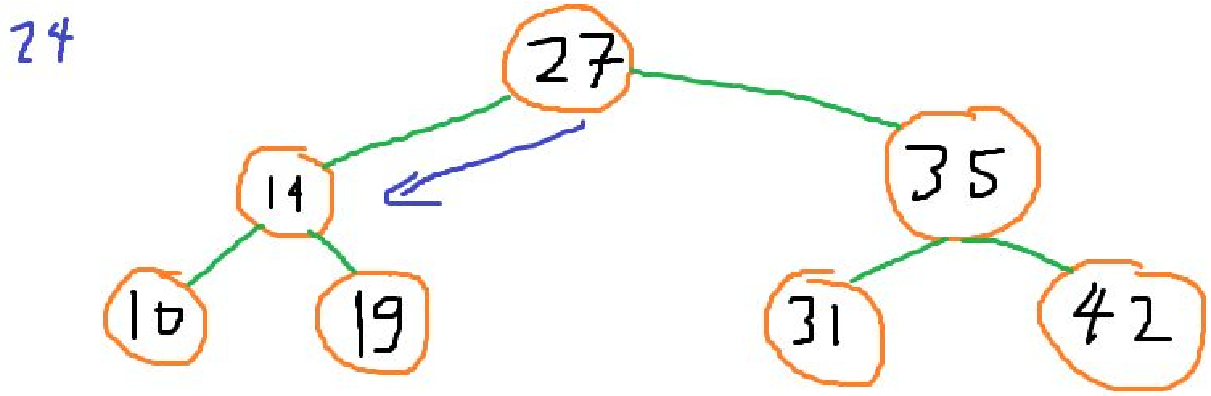
FOR NUMBER 2 AND 3!



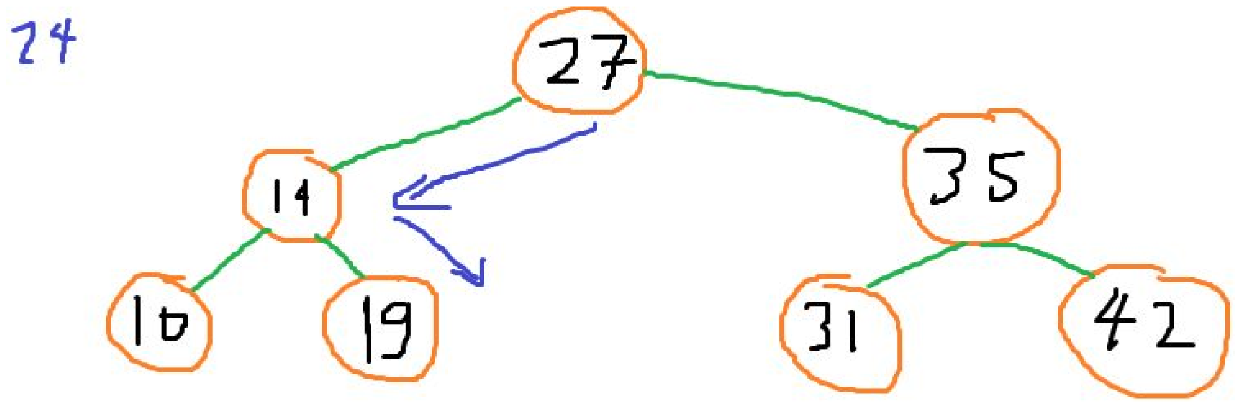
2. **Simulate and explain clearly step by step the process of insertion: 24, 18, 55!**



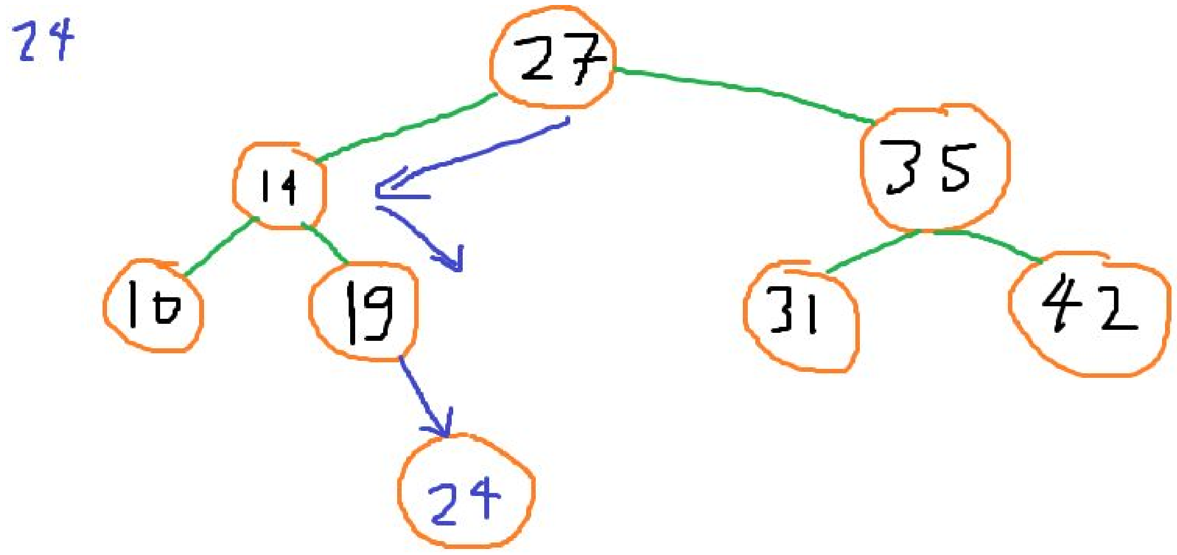
24 is smaller than 27!

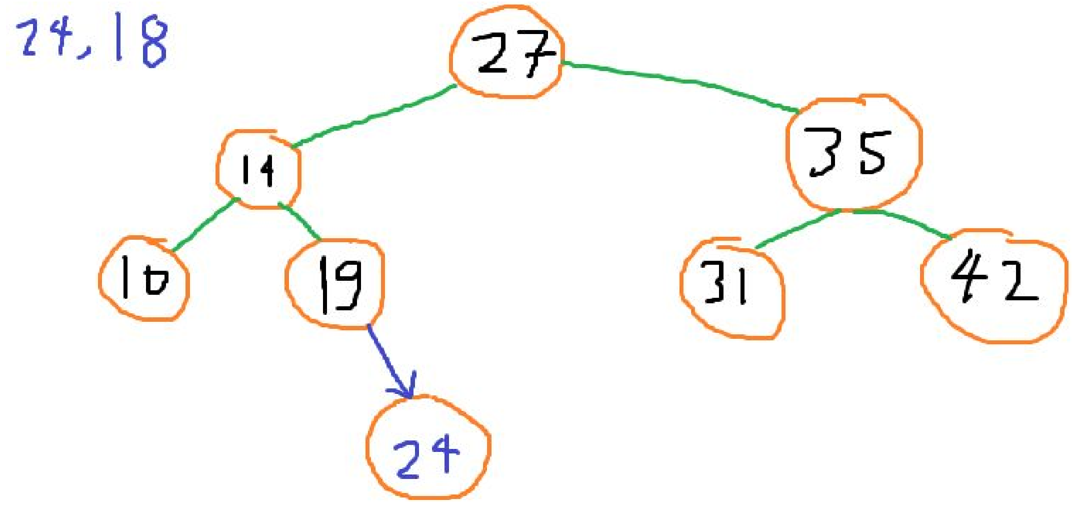


24 is bigger than 14!

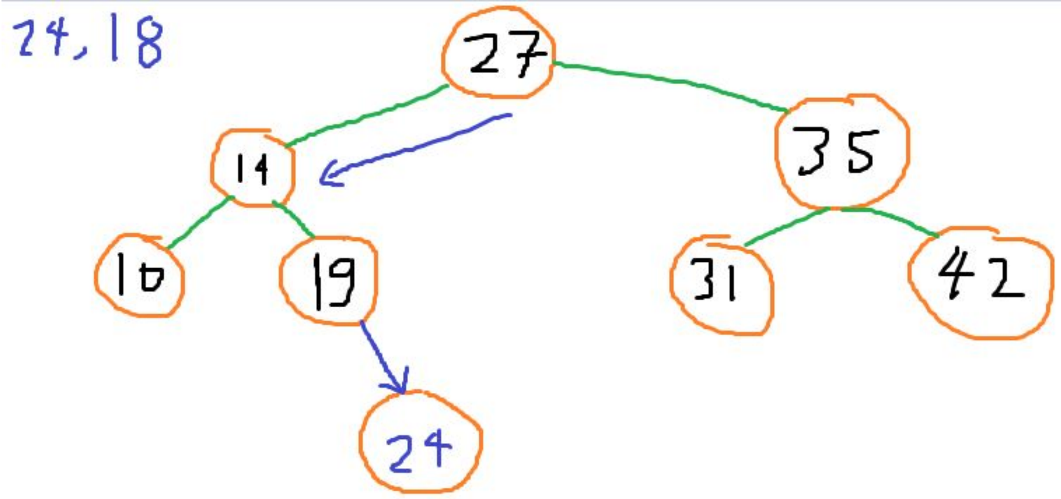


24 is bigger than 19!

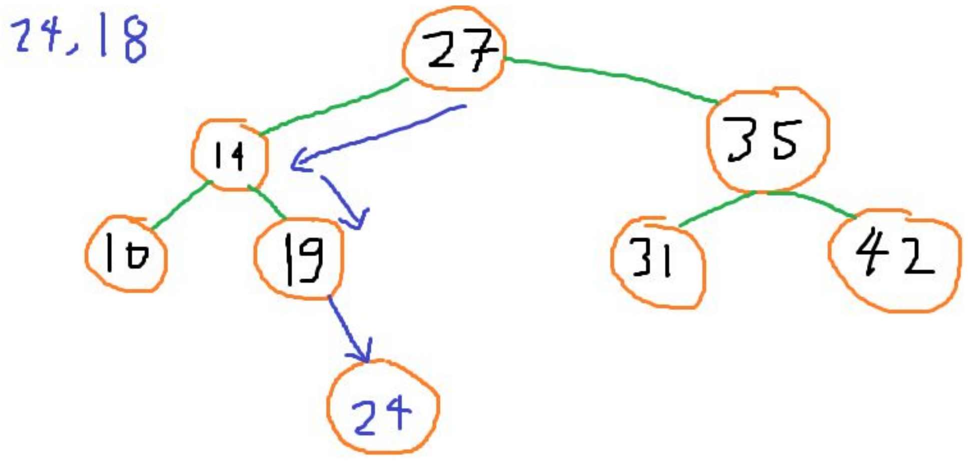




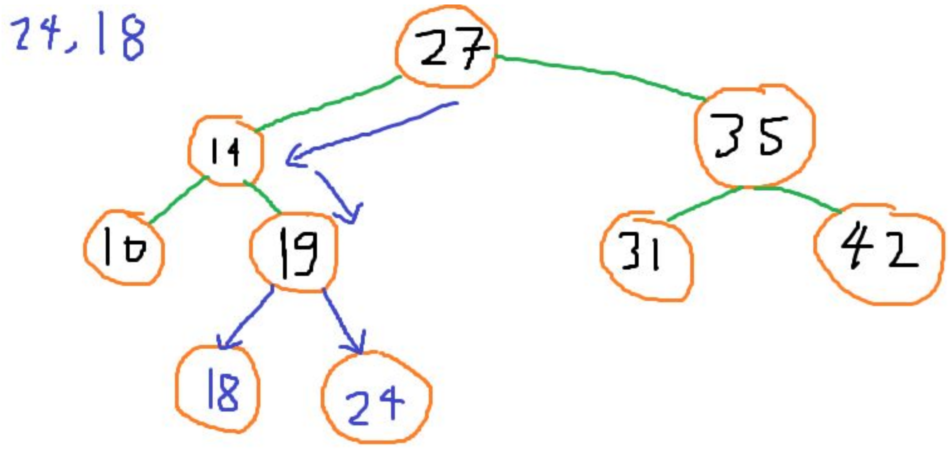
18 is smaller than 27!

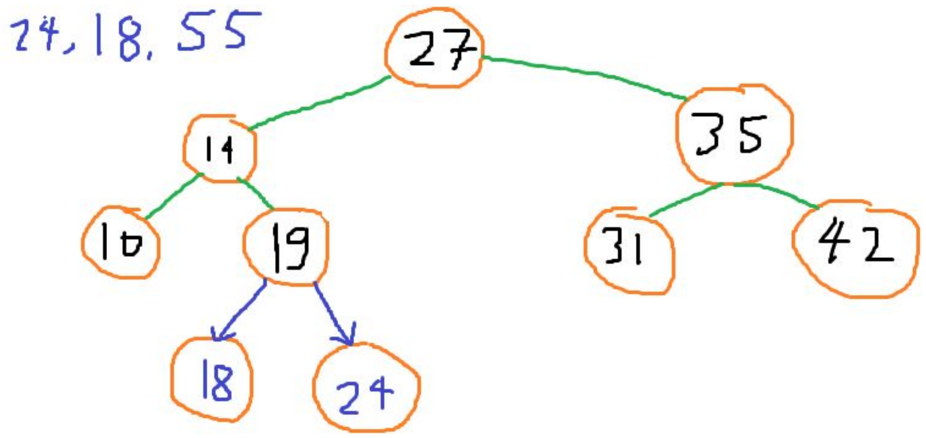


18 is bigger than 14!

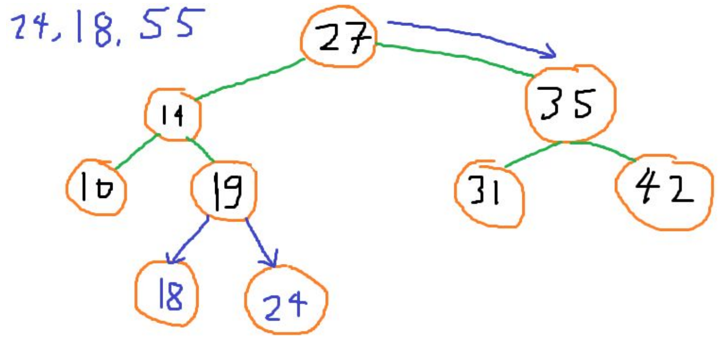


18 is smaller than 19!

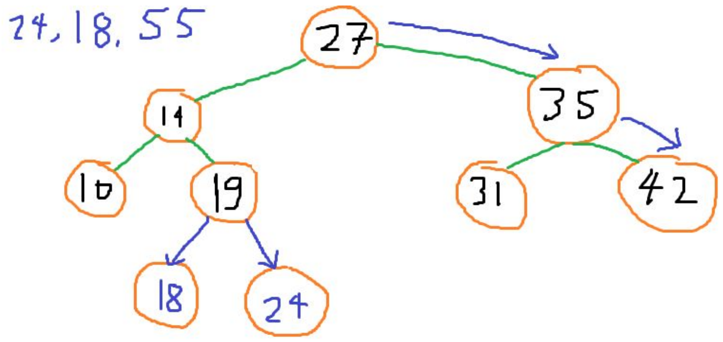




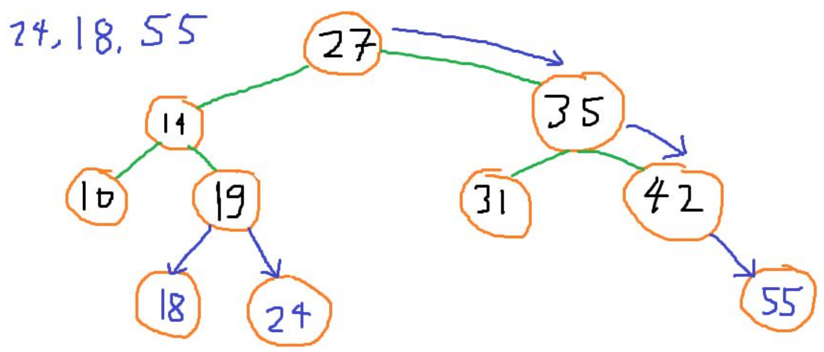
55 is bigger than 27!



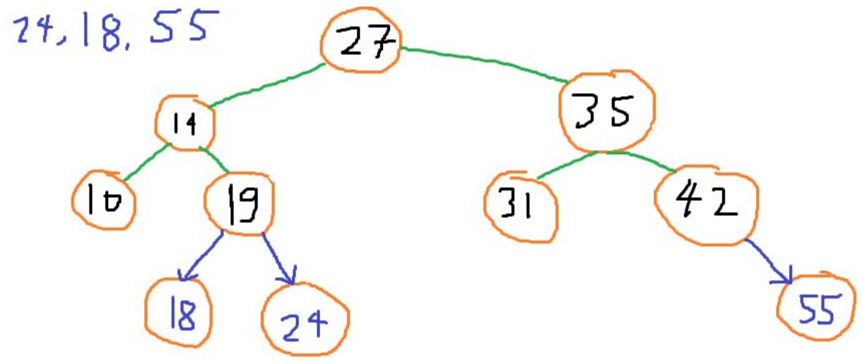
55 is bigger than 35!



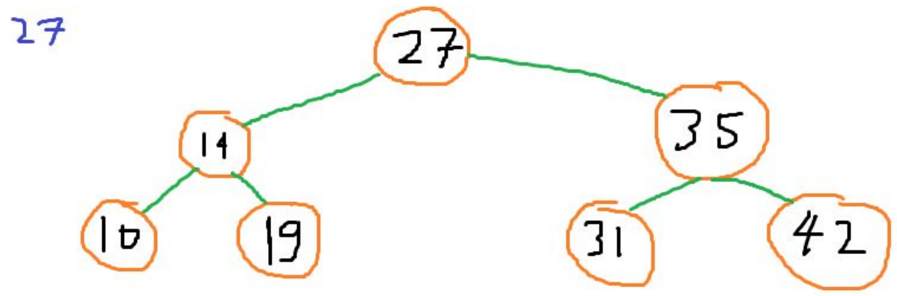
55 is bigger than 42!


24, 18, 55
27
14
35
10
19
31
42
18
24
55

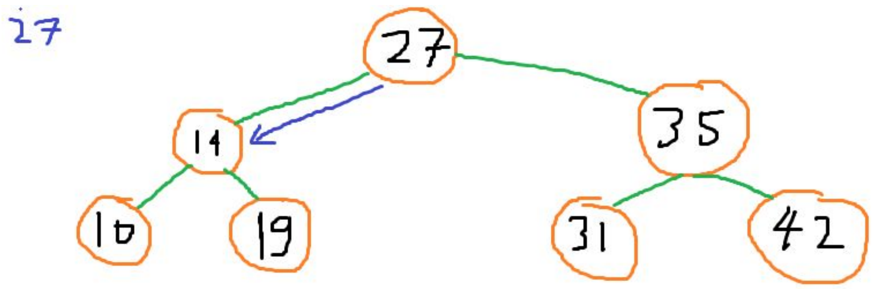
End result:


24, 18, 55
27
14
35
10
19
31
42
18
24
55

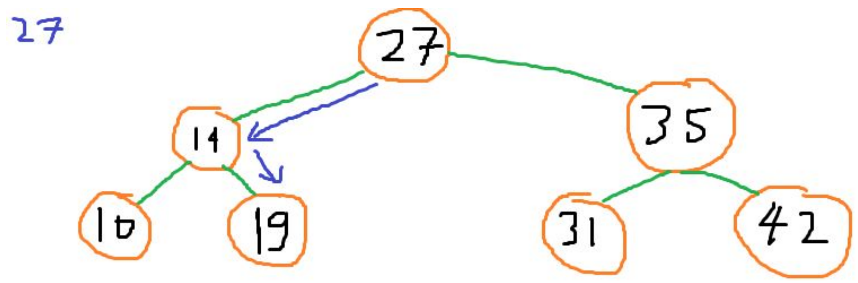
3. **Simulate and explain clearly step by step the process of deletion: 27, 35, 42!**



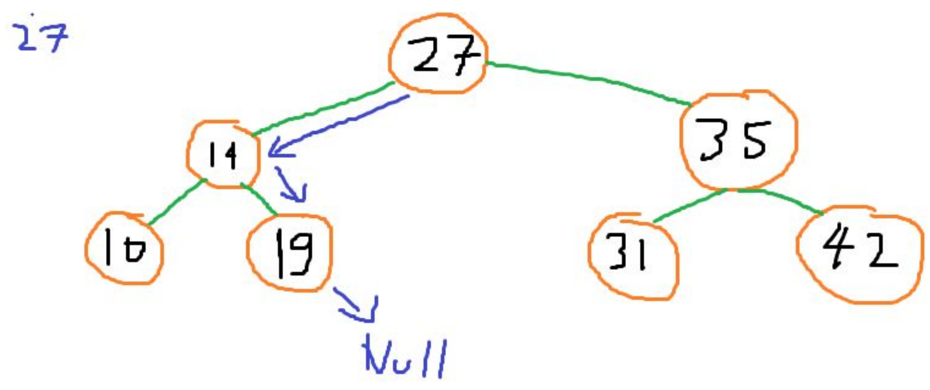
Found 27 at the top! -> go to the left side
Left side = NULL?



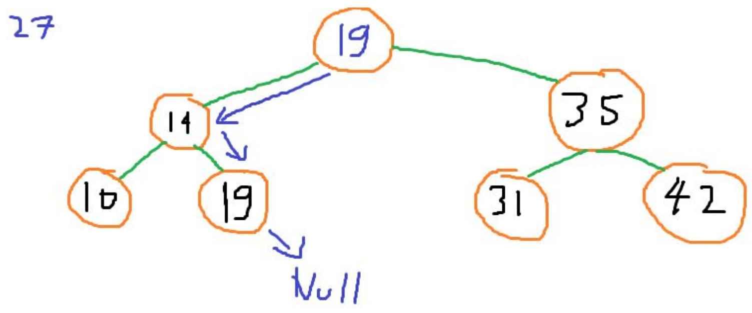
Answer: No! -> go to the right side
Right side = NULL?



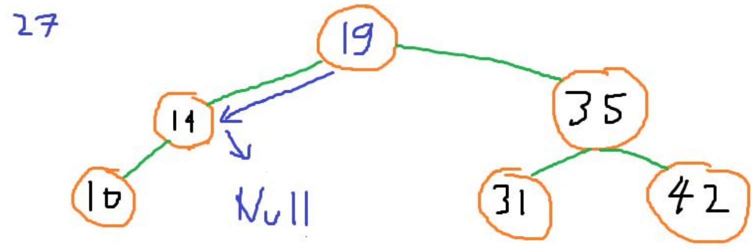
Answer: No! -> keep going to the right
Right side = NULL?

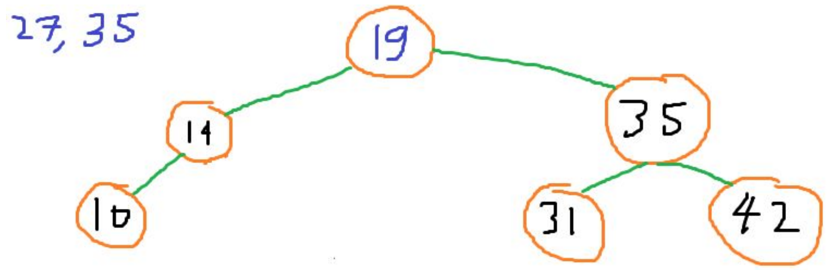


Answer: Yes! -> change the initial node with 19

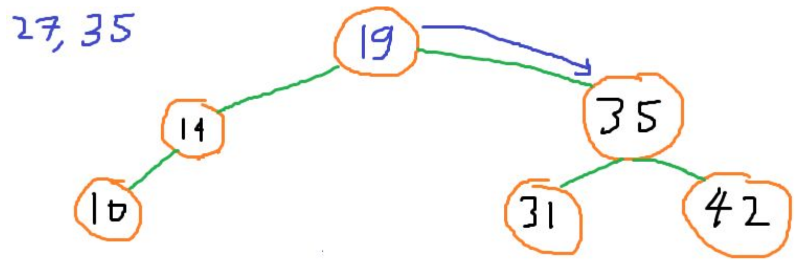


Delete node 19 (the one whose right side is a NULL)

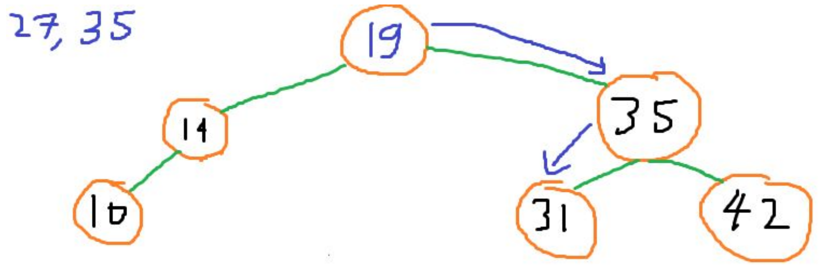




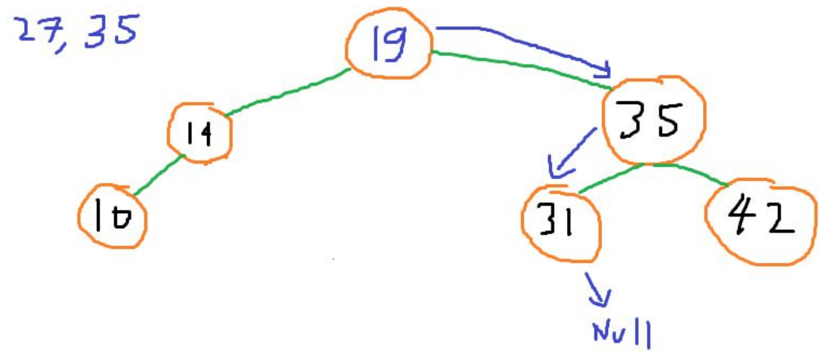
35 is bigger than 19! -> search right side!



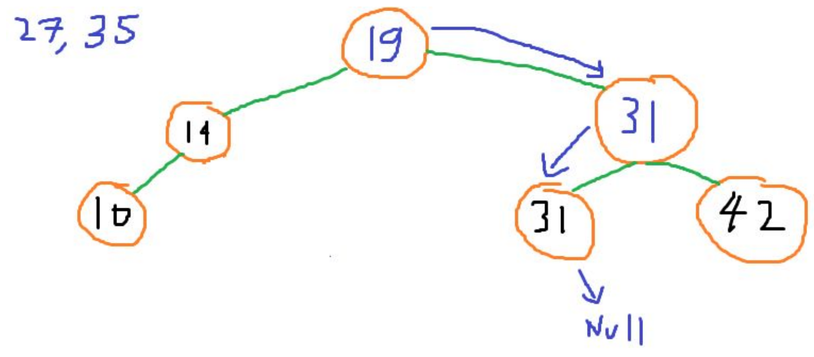
Found 35! -> go to left side
Left side = NULL?



Answer = No! -> go to right side
Right side = NULL?

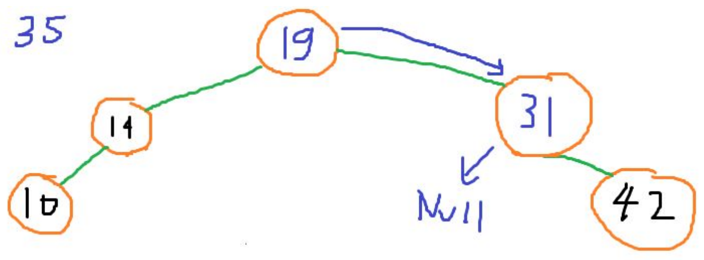


Answer: Yes! -> replace 35 with 31

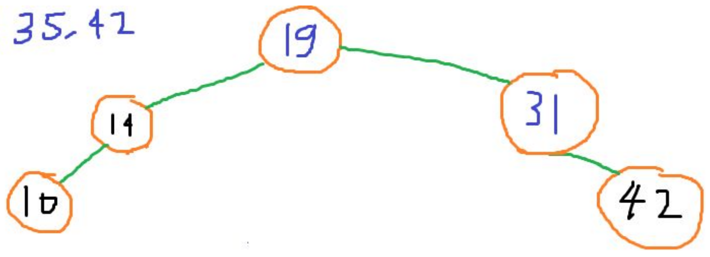


Delete node 31

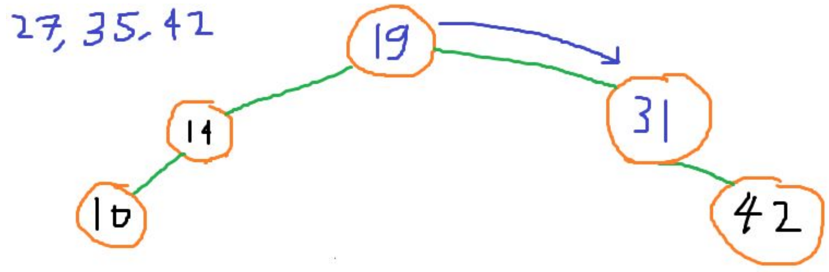
27, 35



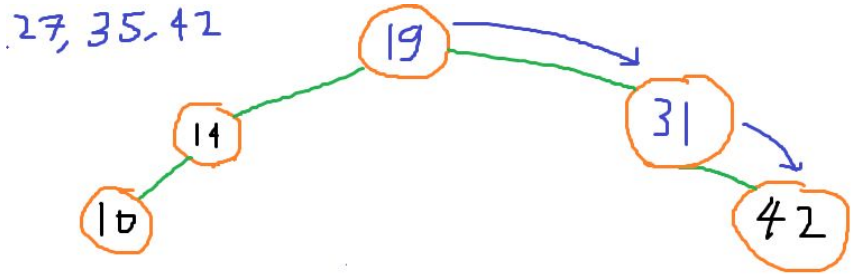
27, 35, 42



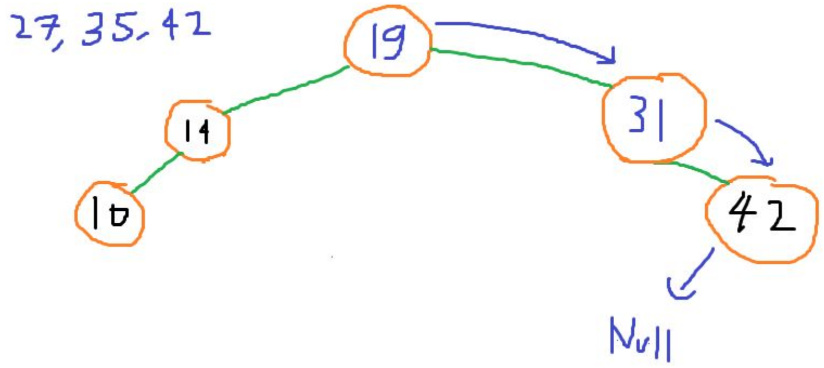
42 is bigger than 19! -> search the right side!



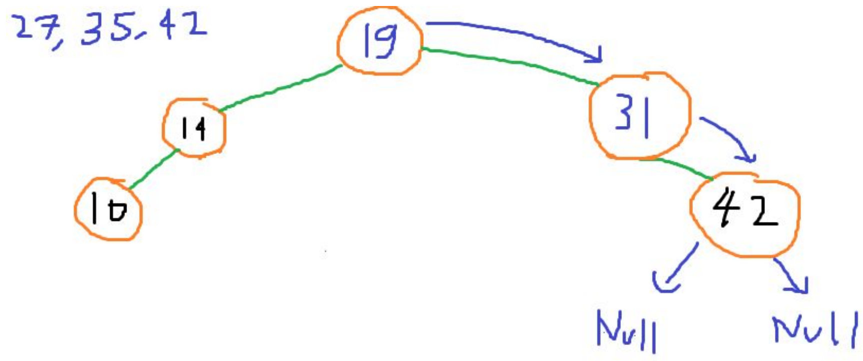
42 is bigger than 31! -> search the right side!



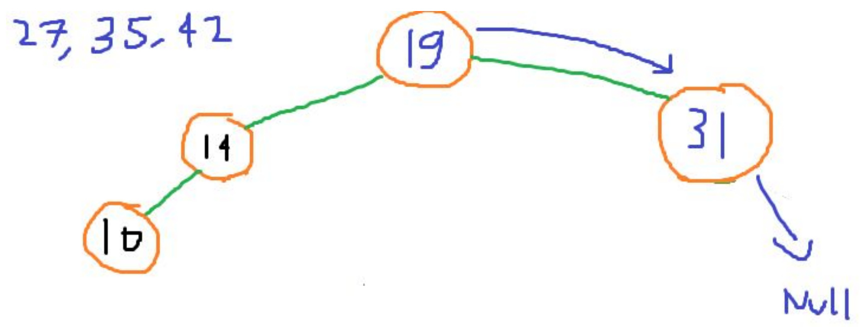
Found 42! Search the left side!
Left side = NULL?



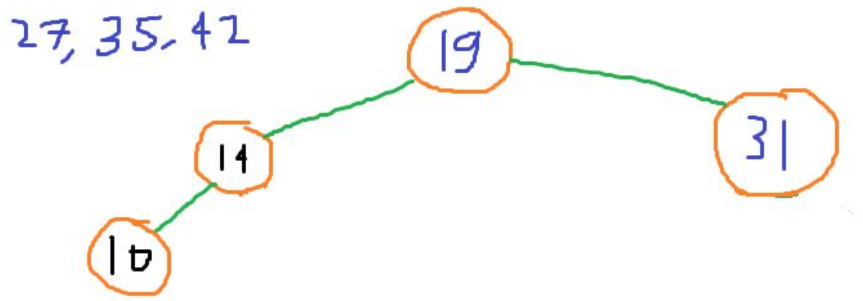
Answer: Yes! Try the right side!
Right side = NULL?



Answer: Yes!
Delete node 42!



End result:


27, 35, 42
19
14
10
31