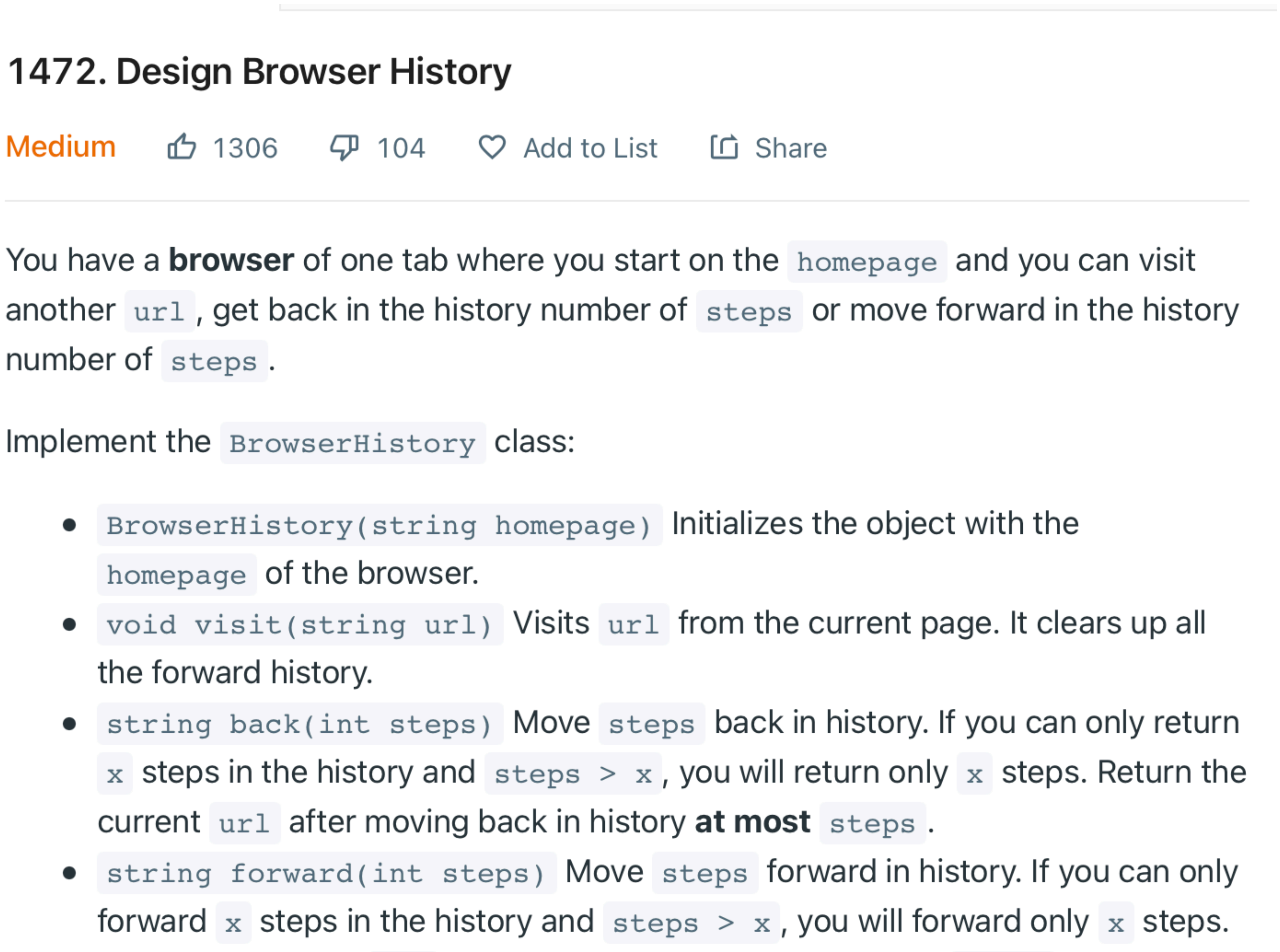


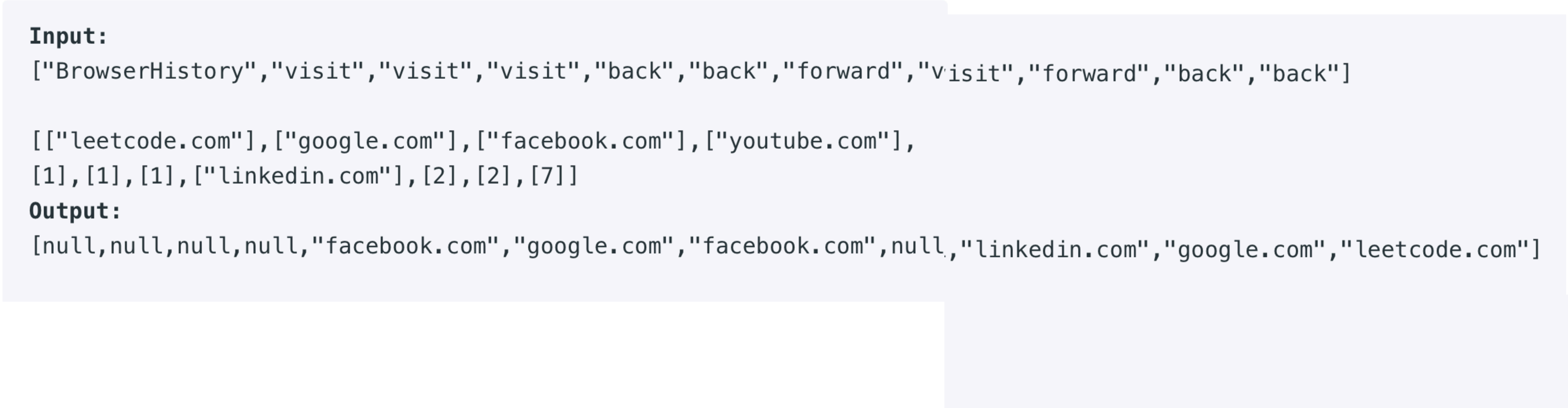
1472. Design Browser History

Medium 👍 1306 👎 104 ♡ Add to List Share

You have a **browser** of one tab where you start on the `homepage` and you can visit another `url`, get back in the history number of `steps` or move forward in the history number of `steps`.

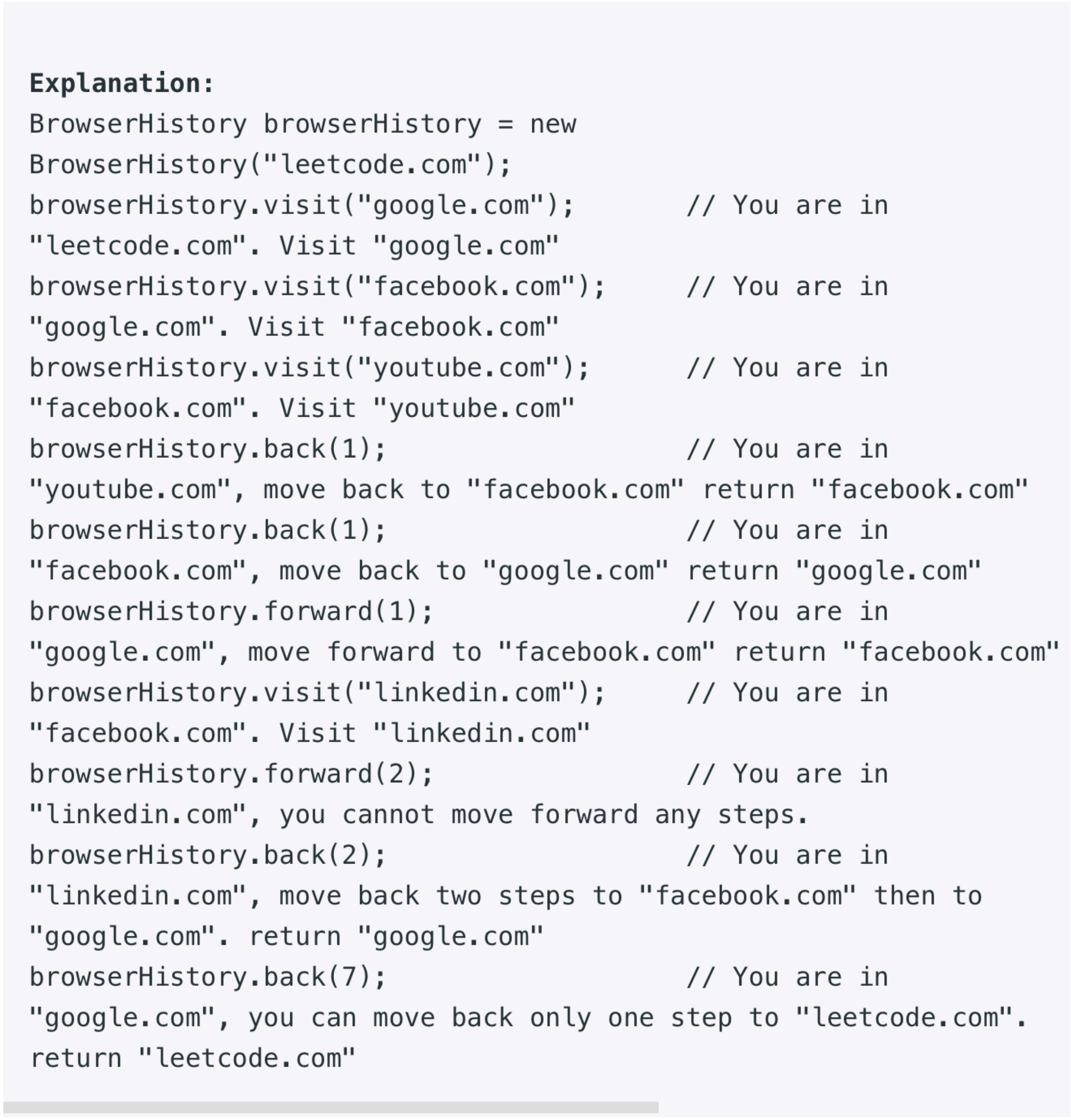
Implement the `BrowserHistory` class:

- `BrowserHistory(string homepage)` Initializes the object with the `homepage` of the browser.
- `void visit(string url)` Visits `url` from the current page. It clears up all the forward history.
- `string back(int steps)` Move `steps` back in history. If you can only return `x` steps in the history and `steps > x`, you will return only `x` steps. Return the current `url` after moving back in history **at most** `steps`.
- `string forward(int steps)` Move `steps` forward in history. If you can only forward `x` steps in the history and `steps > x`, you will forward only `x` steps.

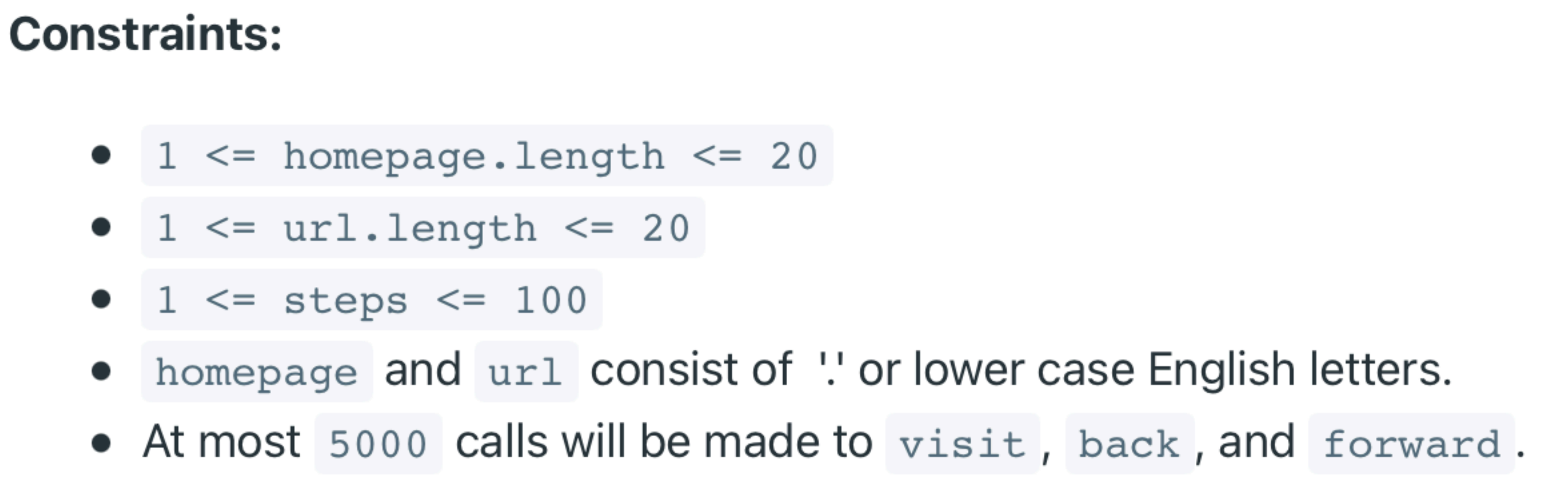


```
Input:
["BrowserHistory","visit","visit","visit","back","back","forward","visit","forward","back","back"]

[["leetcode.com"],["google.com"],["facebook.com"],["youtube.com"],
[1],[1],[1],["linkedin.com"],[2],[2],[7]]
Output:
[null,null,null,null,"facebook.com","google.com","facebook.com",null,"linkedin.com","google.com","leetcode.com"]
```



```
Explanation:
BrowserHistory browserHistory = new
BrowserHistory("leetcode.com");
browserHistory.visit("google.com");       // You are in
"leetcode.com". Visit "google.com"
browserHistory.visit("facebook.com");     // You are in
"google.com". Visit "facebook.com"
browserHistory.visit("youtube.com");      // You are in
"facebook.com". Visit "youtube.com"
browserHistory.back(1);                   // You are in
"youtube.com", move back to "facebook.com" return "facebook.com"
browserHistory.back(1);                   // You are in
"facebook.com", move back to "google.com" return "google.com"
browserHistory.forward(1);                // You are in
"google.com", move forward to "facebook.com" return "facebook.com"
browserHistory.visit("linkedin.com");     // You are in
"facebook.com". Visit "linkedin.com"
browserHistory.forward(2);                // You are in
"linkedin.com", you cannot move forward any steps.
browserHistory.back(2);                   // You are in
"linkedin.com", move back two steps to "facebook.com" then to
"google.com". return "google.com"
browserHistory.back(7);                   // You are in
"google.com", you can move back only one step to "leetcode.com".
return "leetcode.com"
```

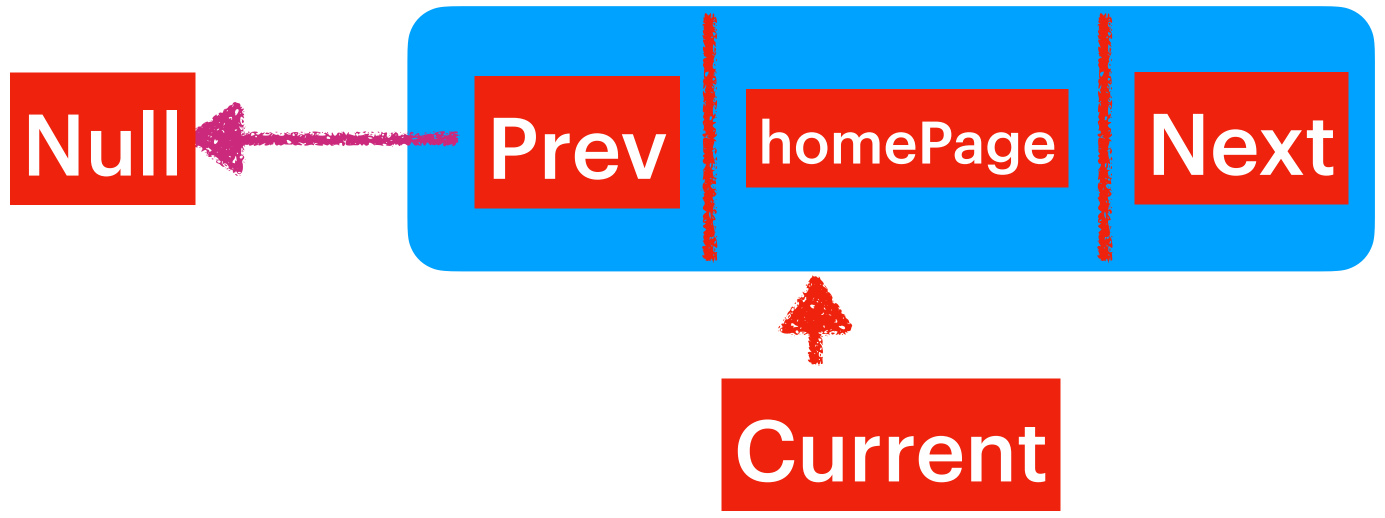


Constraints:

- `1 <= homepage.length <= 20`
- `1 <= url.length <= 20`
- `1 <= steps <= 100`
- `homepage` and `url` consist of '.' or lower case English letters.
- At most `5000` calls will be made to `visit`, `back`, and `forward`.

Algorithm

visit(homePage)



Null

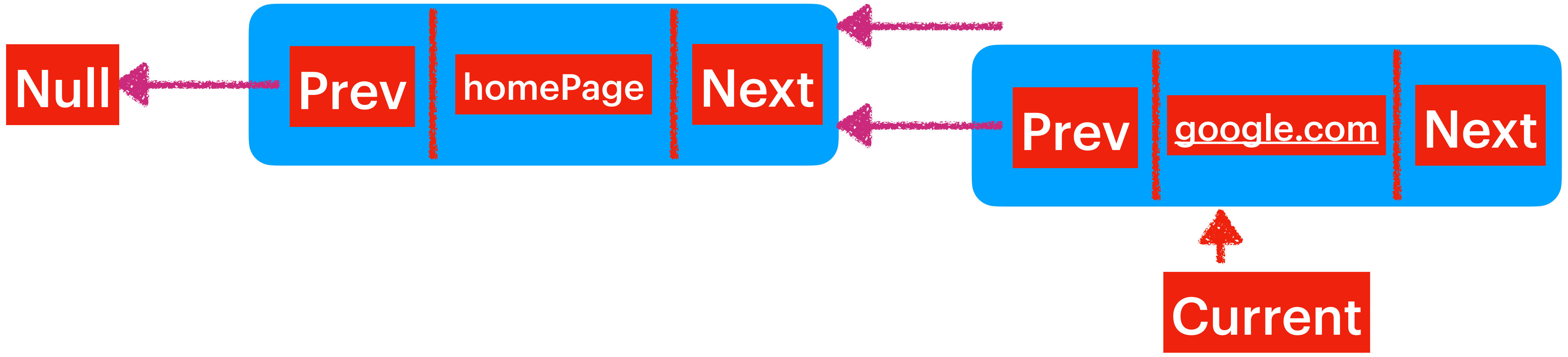
Prev

homePage

Next

Current

visit(google.com);



Null

Prev

homePage

Next

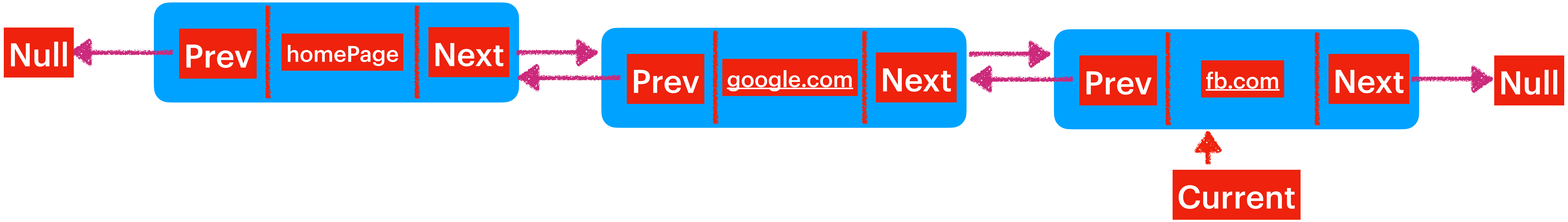
Prev

google.com

Next

Current

visit(fb.com);



Null

Prev

homePage

Next

Prev

google.com

Next

Prev

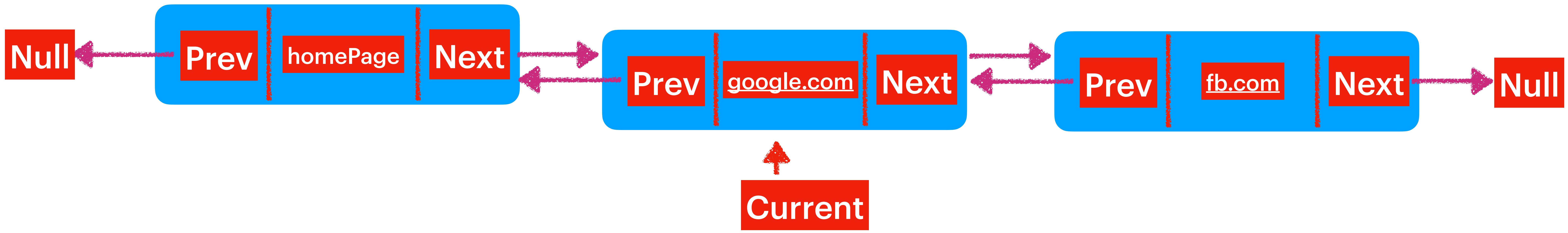
fb.com

Next

Null

Current

back(1) → google.com



Null

Prev

homePage

Next

Prev

google.com

Next

Prev

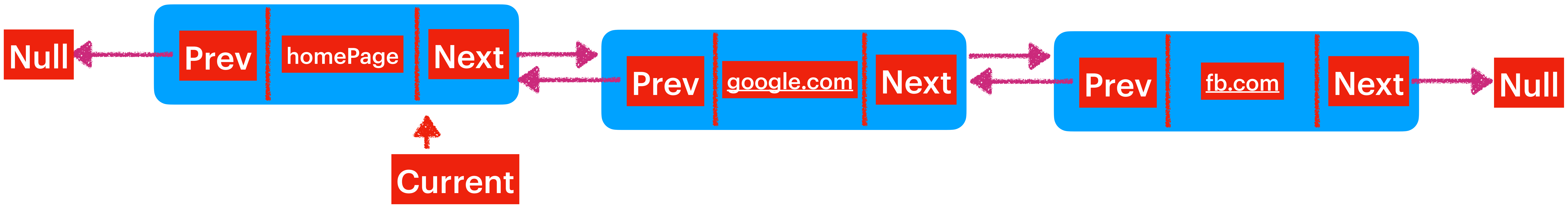
fb.com

Next

Null

Current

back(1) → homePage



Null

Prev

homePage

Next

Prev

google.com

Next

Prev

fb.com

Next

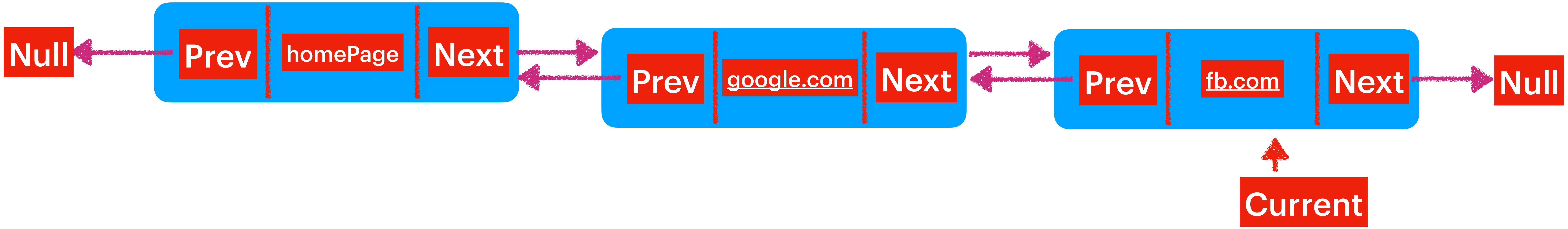
Null

Current

back(3) → homePage

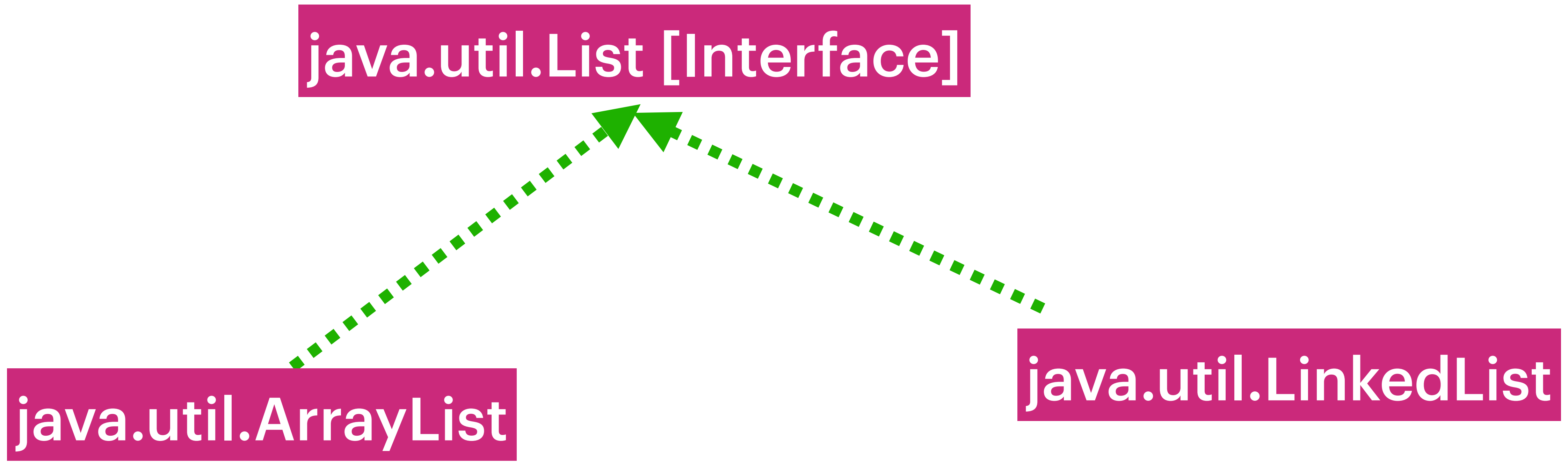
There is no back returns homePage.

forward(2) → fb.com



forward(3) → fb.com

There is no forward returns fb.com.



java.util.List [Interface]

java.util.ArrayList

java.util.LinkedList

Common Properties

Allows duplicates ,
Add takes O(1),
Search takes O(n),
Dynamic in Size.

Specific ArrayList

Delete takes linear search O(n) & linear swaps O(n)
In ArrayList if we know the index then we access the elements in constant time : O(1)

Specific LinkedList

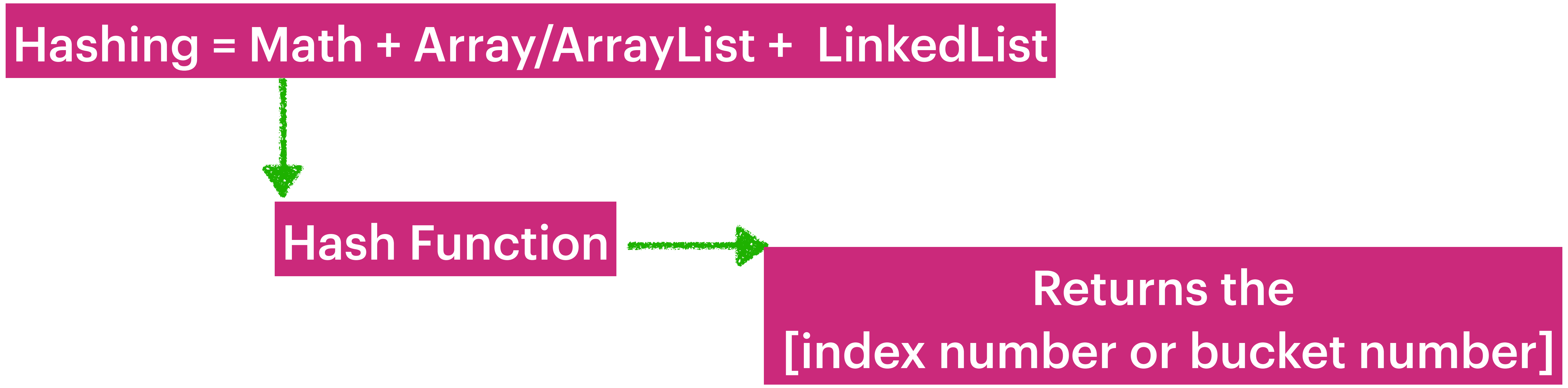
In Java the LinkedList is DoubleLinkedList.
Delete takes linear search O(n) & Constant swap O(1)
In LinkedList if even we know the index takes linear time to get the element : O(n)
As in the LinkedList the memory blocks are not continuous .

Why Hashing ?

With Hashing [We can Achieve]
search : O(1)
add : O(1)
delete : O(1)
At the cost of hashing we avoid duplicates.

Array/ ArrayList :
If we know the index access is O(1).

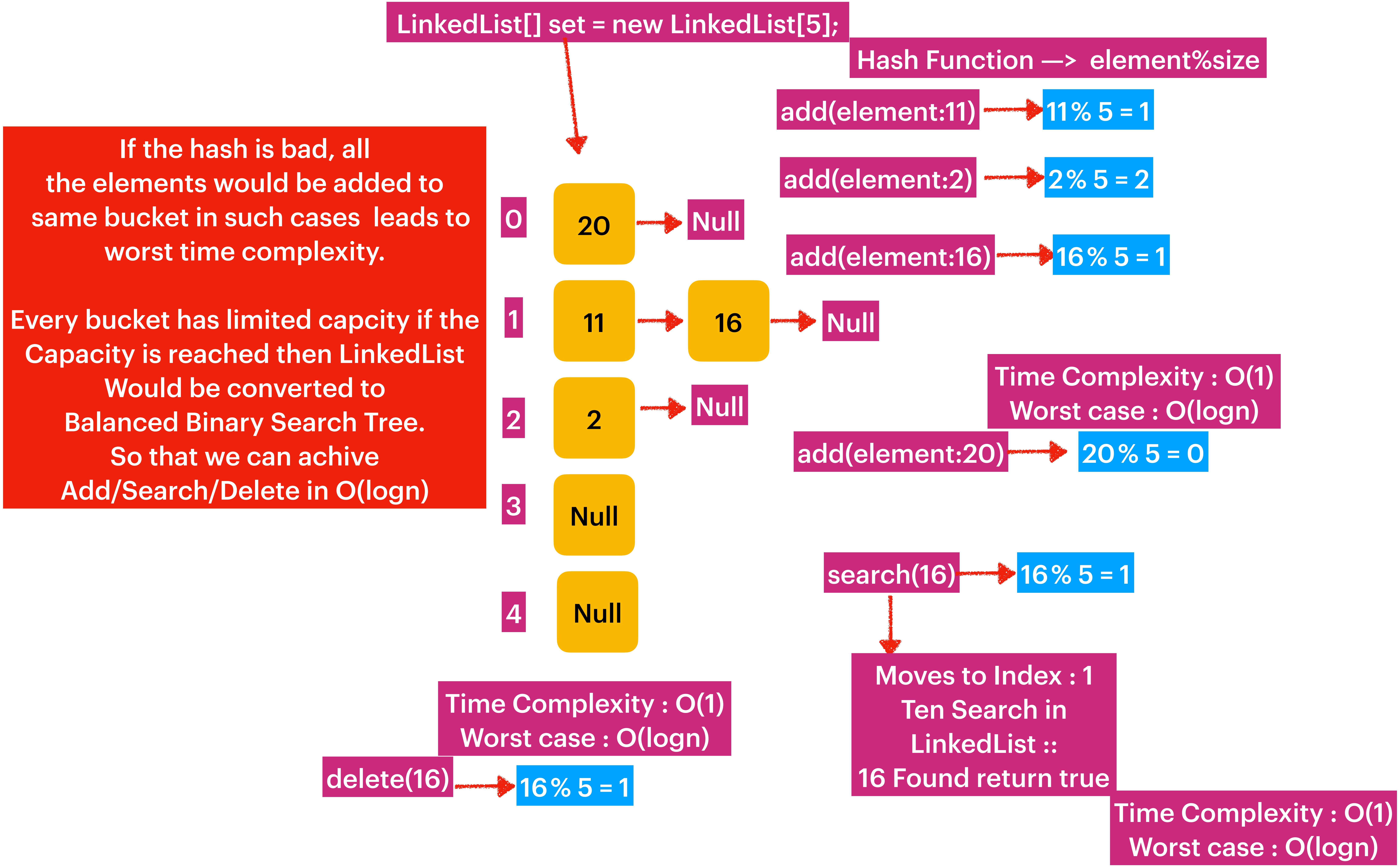
LinkedList: Delete Operation Swap is O(1) , Add in the Middle is O(1).



Hashing = Math + Array/ArrayList + LinkedList

Hash Function

Returns the
[index number or bucket number]



LinkedList[] set = new LinkedList[5];

Hash Function —> element%size

add(element:11) → 11% 5 = 1

add(element:2) → 2% 5 = 2

add(element:16) → 16 % 5 = 1

add(element:20) → 20 % 5 = 0

Time Complexity : O(1)
Worst case : O(logn)

If the hash is bad, all the elements would be added to same bucket in such cases leads to worst time complexity.

Every bucket has limited capcity if the Capacity is reached then LinkedList Would be converted to Balanced Binary Search Tree. So that we can achive Add/Search/Delete in O(logn)

search(16) → 16 % 5 = 1

Moves to Index : 1
Ten Search in
LinkedList ::
16 Found return true

Time Complexity : O(1)
Worst case : O(logn)

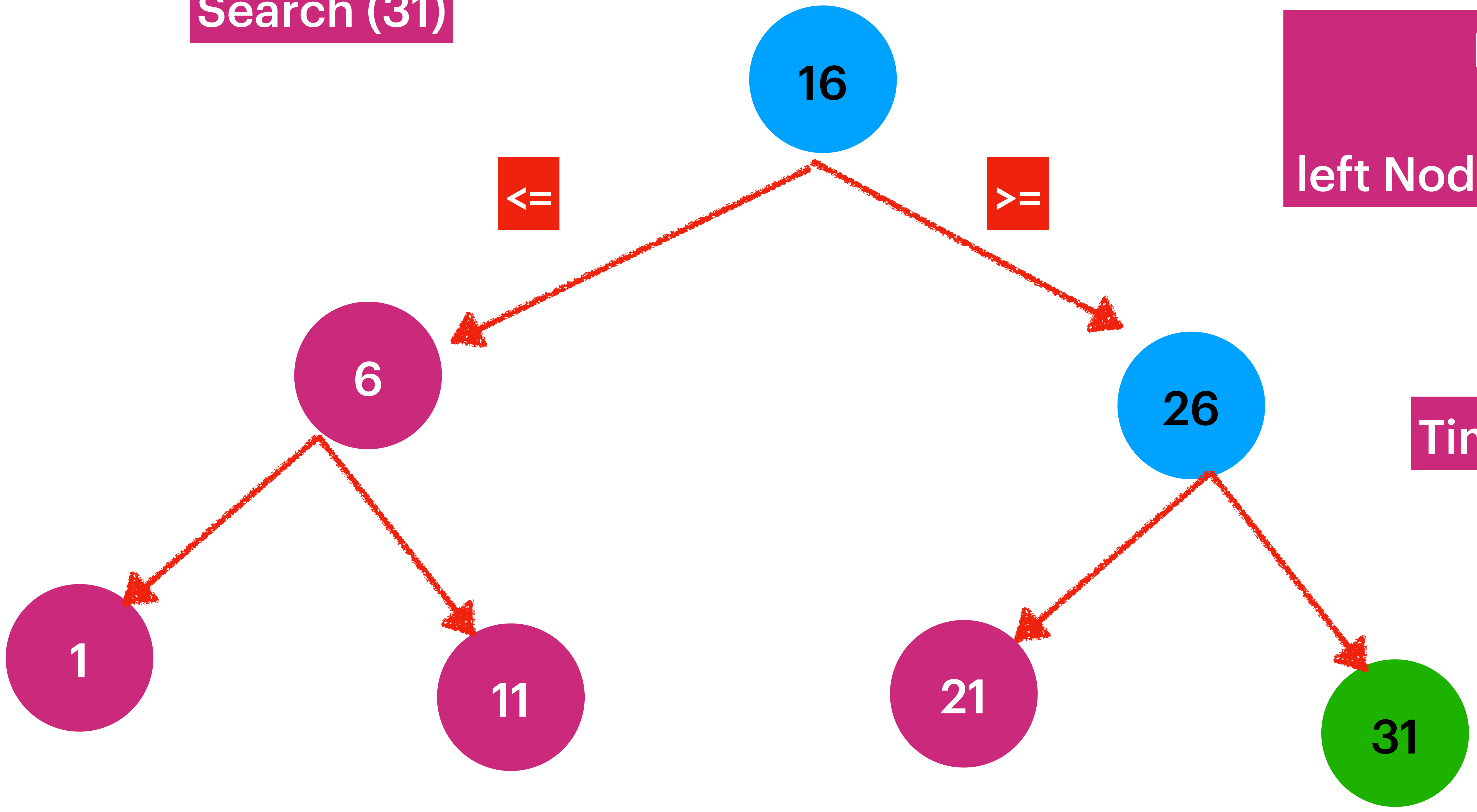
Time Complexity : O(1)
Worst case : O(logn)

delete(16) → 16 % 5 = 1

1 -> 6 -> 11 -> 16 -> 21 -> 26 -> 31

When we have bad hash then there is possibility that
All the elements mapped same bucket which causes O(n) in search/delete operation,
To avoid this we will have hieight balanced binery search tree when the size >= capacity

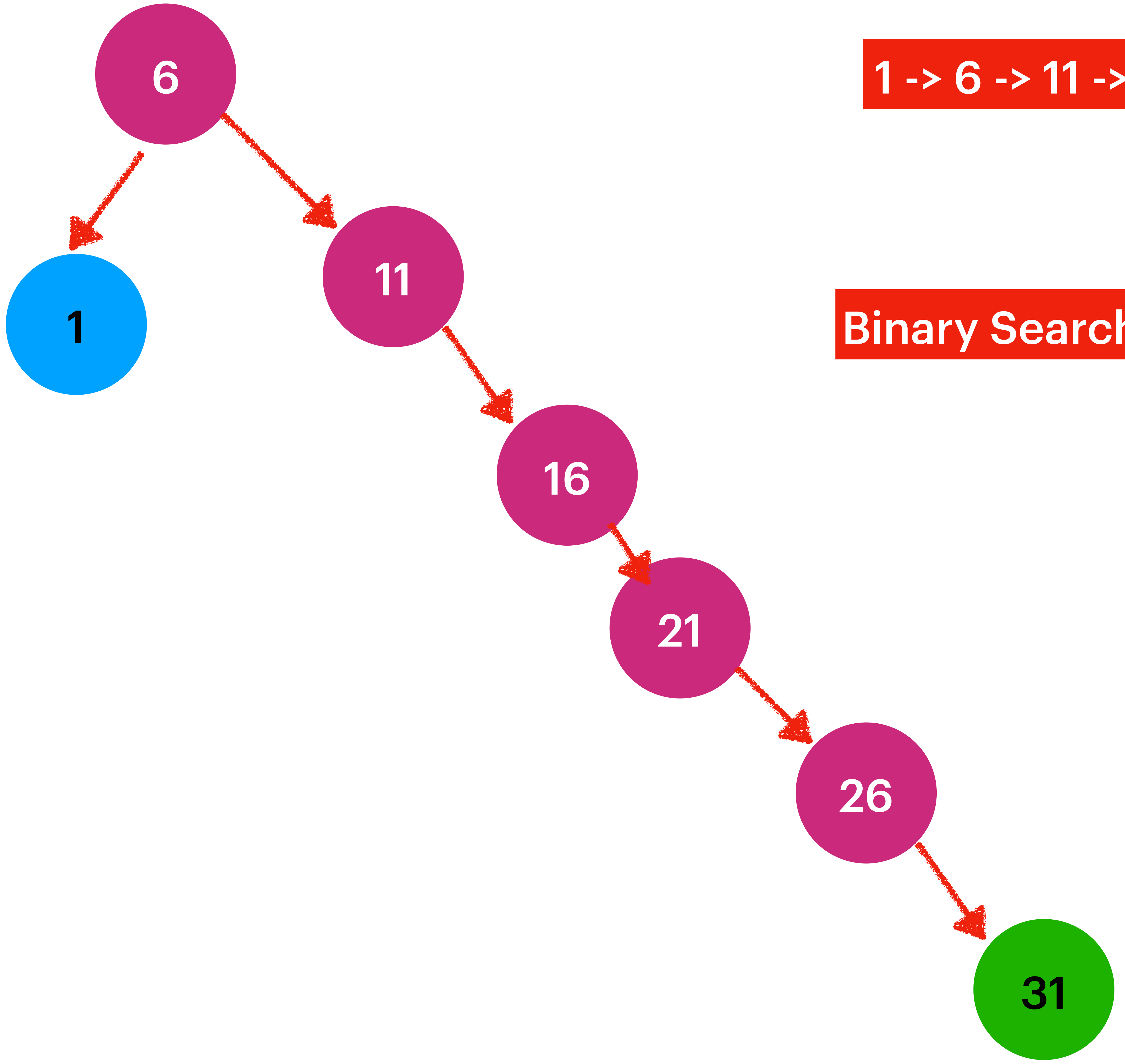
Search (31)



Binary Search Tree : Take any
Node :
left Node value are <= & Right Node values >=

Time Complexity : O(logn)

Height Balanced Binary Search Tree :
Its a Binary Search Tree, the max height difference between left sub tree and right sub tree is 1.



1 -> 6 -> 11 -> 16 -> 21 -> 26 -> 31

Binary Search Tree but not height balanced.

Search(31)

In Worst case it becomes : O(n)