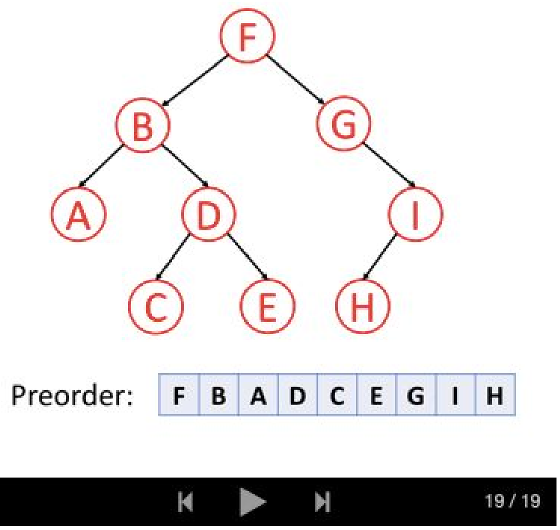
Pre-order Traversal

Pre-order traversal is to visit the root first. Then traverse the left subtree. Finally, traverse the right subtree.

Here is an example:




```python
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
        # recursively
    def preorderTraversal1(self, root):
        res = []
        self.dfs(root, res)
        return res

    def dfs(self, root, res):
        if root:
            res.append(root.val)
            self.dfs(root.left, res)
            self.dfs(root.right, res)

    # iteratively
    def preorderTraversal(self, root):
        stack, res = [root], []
        while stack:
            node = stack.pop()
            if node:
                res.append(node.val)
                stack.append(node.right)
                stack.append(node.left)
        return res
```

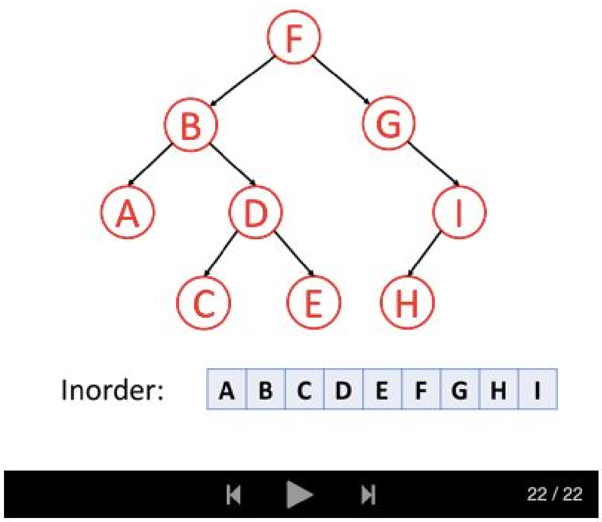
Custom Testcase (Contribute)

Run Code   Submit Solution

In-order Traversal

In-order traversal is to traverse the left subtree first. Then visit the root. Finally, traverse the right subtree.

Let's do in-order traversal together:



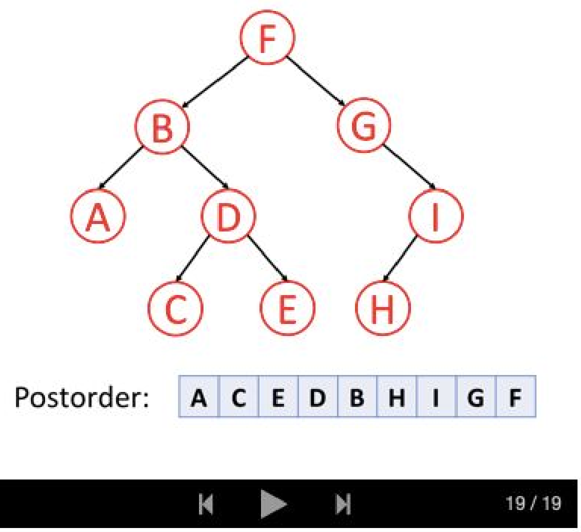
Typically, for `binary search tree`, we can retrieve all the data in sorted order using in-order traversal. We will mention that again in another card(Introduction to Data Structure - Binary Search Tree).

```python
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    # recursively
    def inorderTraversal1(self, root):
        res = []
        self.helper(root, res)
        return res

    def helper(self, root, res):
        if root:
            self.helper(root.left, res)
            res.append(root.val)
            self.helper(root.right, res)

    # iteratively
    def inorderTraversal(self, root):
        res, stack = [], []
        while True:
            while root:
                stack.append(root)
                root = root.left
            if not stack:
                return res
            node = stack.pop()
            res.append(node.val)
            root = node.right
```

Post-order Traversal

Post-order traversal is to traverse the left subtree first. Then traverse the right subtree. Finally, visit the root.

Here is an animation to help you understand post-order traversal:



It is worth noting that when you delete nodes in a tree, deletion process will be in post-order. That is to say, when you delete a node, you will delete its left child and its right child before you delete the node itself.

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
        # recursively
    def postorderTraversal1(self, root):
        res = []
        self.dfs(root, res)
        return res

    def dfs(self, root, res):
        if root:
            self.dfs(root.left, res)
            self.dfs(root.right, res)
            res.append(root.val)

    # iteratively
    def postorderTraversal(self, root):
        res, stack = [], [root]
        while stack:
            node = stack.pop()
            if node:
                res.append(node.val)
                stack.append(node.left)
                stack.append(node.right)
        return res[::-1]
```

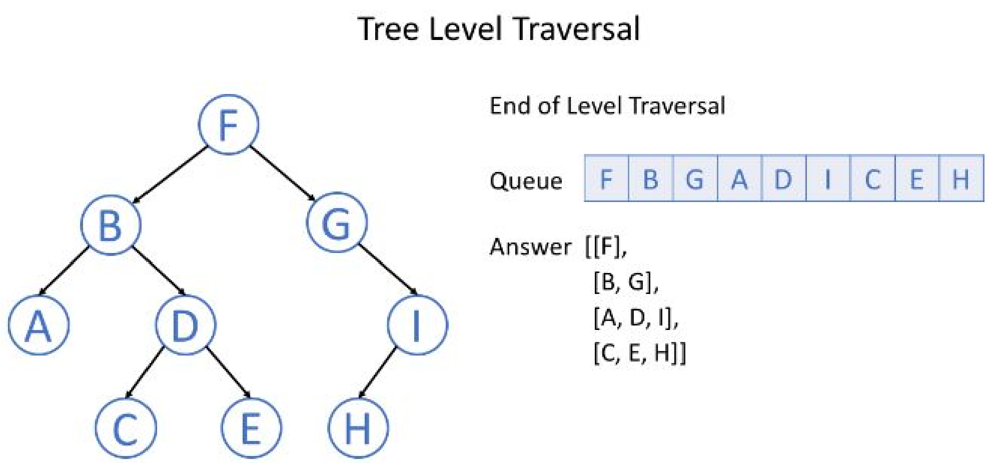
Level-order Traversal - Introduction

Level-order traversal is to traverse the tree level by level.

`Breadth-First Search` is an algorithm to traverse or search in data structures like a tree or a graph. The algorithm starts with a root node and visit the node itself first. Then traverse its neighbors, traverse its second level neighbors, traverse its third level neighbors, so on and so forth.

When we do breadth-first search in a tree, the order of the nodes we visited is in level order.

Here is an example of level-order traversal:

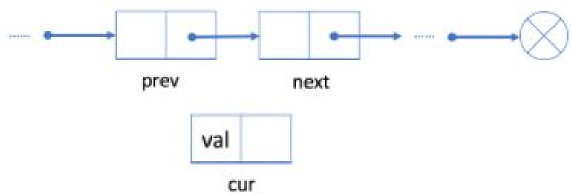



```python
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
class Solution(object):
    def levelOrder(self, root):
        res = []
        self.dfs(root, 0, res)
        return res

    def dfs(self, root, level, res):
        if not root:
            return
        if len(res) < level+1:
            res.append([])
        res[level].append(root.val)
        self.dfs(root.left, level+1, res)
        self.dfs(root.right, level+1, res)
```

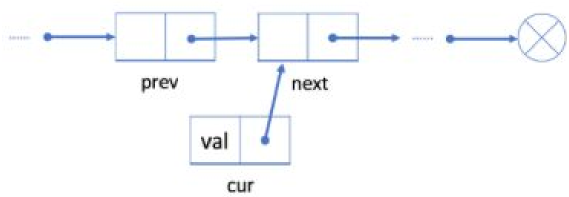
Add Operation - Singly Linked List

If we want to add a new value after a given node `prev`, we should:

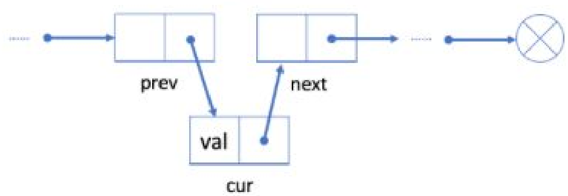
1. Initialize a new node `cur` with the given value;



2. Link the "next" field of `cur` to prev's next node `next`;



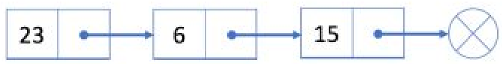
3. Link the "next" field in `prev` to `cur`.



Unlike an array, we don’t need to move all elements past the inserted element. Therefore, you can insert a new node into a linked list in `O(1)` time complexity, which is very efficient.

```python
    def addAtIndex(self, index, val):
        """
        Add a node of value val before the index-th node in the linked list.
        If index equals to the length of linked list, the node will be appended to the end of linked list.
        If index is greater than the length, the node will not be inserted.
        :type index: int
        :type val: int
        :rtype: void
        """
        if index < 0 or index > self.size:
            return

        if index == 0:
            self.addAtHead(val)
        else:
            curr = self.head
            for i in range(index - 1):
                curr = curr.next
            node = Node(val)
            node.next = curr.next
            curr.next = node

            self.size += 1
```

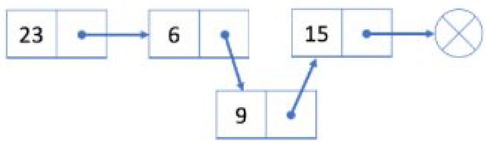
An Example



Let's insert a new value 9 after the second node 6.

We will first initialize a new node with value 9. Then link node 9 to node 15. Finally, link node 6 to node 9.

After insertion, our linked list will look like this:



Add a Node at the Beginning

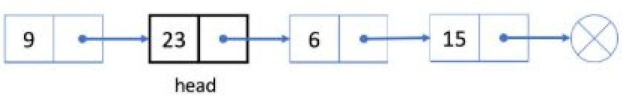
As we know, we use the head node `head` to represent the whole list.

So it is essential to update `head` when adding a new node at the beginning of the list.

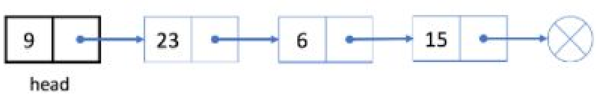
1. Initialize a new node `cur` ;
2. Link the new node to our original head node `head` .
3. Assign `cur` to `head` .

For example, let's add a new node 9 at the beginning of the list.

1. We initialize a new node 9 and link node 9 to current head node 23.



2. Assign node 9 to be our new head.



> What about adding a new node at the end of the list? Can we still use similar strategy?

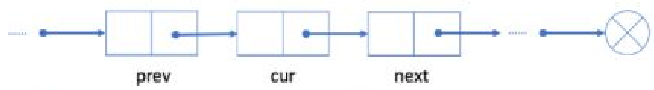
```python
    def addAtHead(self, val):
        """
        Add a node of value val before the first element of the linked list.
        After the insertion, the new node will be the first node of the linked list.
        :type val: int
        :rtype: void
        """
        node = Node(val)
        node.next = self.head
        self.head = node

        self.size += 1
```

```python
    def addAtTail(self, val):
        """
        Append a node of value val to the last element of the linked list.
        :type val: int
        :rtype: void
        """
        curr = self.head
        if curr is None:
            self.head = Node(val)
        else:
            while curr.next is not None:
                curr = curr.next
            curr.next = Node(val)

        self.size += 1
```

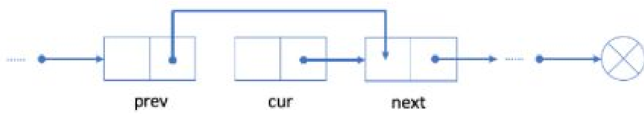
Delete Operation - Singly Linked List

If we want to delete an existing node `cur` from the singly linked list, we can do it in two steps:

1. Find cur's previous node `prev` and its next node `next`;



2. Link `prev` to cur's next node `next`.

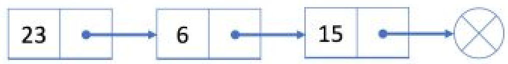


In our first step, we need to find out `prev` and `next`. It is easy to find out `next` using the reference field of `cur`. However, we have to traverse the linked list from the head node to find out `prev` which will take `O(N)` time on average, where N is the length of the linked list. So the time complexity of deleting a node will be `O(N)`.

The space complexity is `O(1)` because we only need constant space to store our pointers.

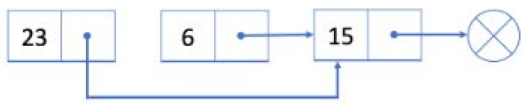
```python
    def deleteAtIndex(self, index):
        """
        Delete the index-th node in the linked list, if the index is valid.
        :type index: int
        :rtype: void
        """
        if index < 0 or index >= self.size:
            return

        curr = self.head
        if index == 0:
            self.head = curr.next
        else:
            for i in range(index - 1):
                curr = curr.next
            curr.next = curr.next.next

        self.size -= 1

```

An Example



Let's try to delete node 6 from the singly linked list above.

1. Traverse the linked list from the head until we find the previous node `prev` which is node 23
2. Link `prev` (node 23) with `next` (node 15)

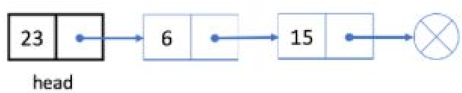


Node 6 is not in our singly linked list now.

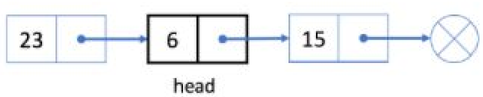
Delete the First Node

If we want to delete the first node, the strategy will be a little different.

As we mentioned before, we use the head node `head` to represent a linked list. Our head is the black node 23 in the example below.



If we want to delete the first node, we can simply `assign the next node to head`. That is to say, our head will be node 6 after deletion.



The linked list begins at the head node, so node 23 is no longer in our linked list.

> What about deleting the last node? Can we still use similar strategy?

Reverse Linked List

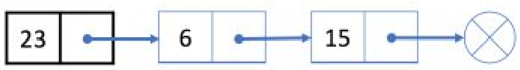
Let's start with a classic problem:

> Reverse a singly linked list.

One solution is to `iterate the nodes in original order and move them to the head of the list one by one`. It seems hard to understand. We will first use an example to go through our algorithm.

Algorithm Overview

Let's look at an example:



Keep in mind that the black node 23 is our original head node.

1. First, we move the next node of the black node, which is node 6, to the head of the list:

2. Then we move the next node of the black node, which is node 15, to the head of the list:

3. The next node of the black node now is null. So we stop and return our new head node 15.

More

In this algorithm, each node will be moved `exactly once`.

Therefore, the time complexity is `O(N)`, where N is the length of the linked list. We only use constant extra space so the space complexity is `O(1)`.

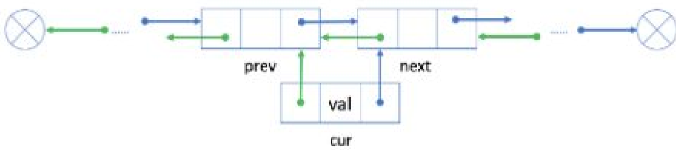
This problem is the foundation of many linked-list problems you might come across in your interview. If you are still stuck, our next article will talk more about the implementation details.

There are also many other solutions. You should be familiar with at least one solution and be able to implement it.

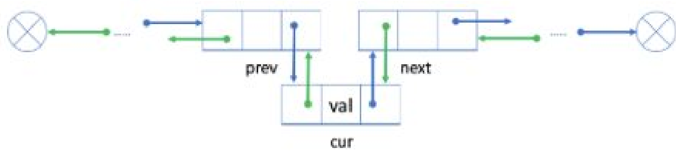
Add Operation - Doubly Linked List

If we want to insert a new node `cur` after an existing node `prev`, we can divide this process into two steps:

1. link `cur` with `prev` and `next`, where `next` is the original next node of `prev`;



2. re-link the `prev` and `next` with `cur`.

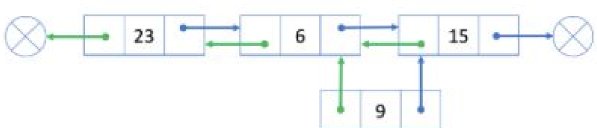


Similar to the singly linked list, both the time and the space complexity of the add operation are `O(1)`.

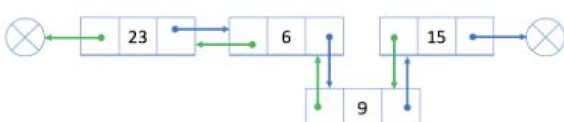
An Example

Let's add a new node 9 after the existing node 6:

1. link `cur` (node 9) with `prev` (node 6) and `next` (node 15)



2. re-link `prev` (node 6) and `next` (node 15) with `cur` (node 9)



> What if we want to insert a new node `at the beginning` or `at the end`?

```python
def addAtHead(self, val):
    """
    Add a node of value val before the first element of the linked list.
    After the insertion, the new node will be the first node of the linked list.
    :type val: int
    :rtype: void
    """
    node = Node(val)
    node.next = self.head
    self.head = node

    self.size += 1
```

```python
def addAtTail(self, val):
    """
    Append a node of value val to the last element of the linked list.
    :type val: int
    :rtype: void
    """
    curr = self.head
    if curr is None:
        self.head = Node(val)
    else:
        while curr.next is not None:
            curr = curr.next
        curr.next = Node(val)

    self.size += 1
```

```python
def addAtIndex(self, index, val):
    """
    Add a node of value val before the index-th node in the linked list.
    If index equals to the length of linked list, the node will be appended to the end of linked list.
    If index is greater than the length, the node will not be inserted.
    :type index: int
    :type val: int
    :rtype: void
    """
    if index < 0 or index > self.size:
        return

    if index == 0:
        self.addAtHead(val)
    else:
        curr = self.head
        for i in range(index - 1):
            curr = curr.next
        node = Node(val)
        node.next = curr.next
        curr.next = node

        self.size += 1
```

A Delete Operation - Doubly Linked List

If we want to delete an existing node `cur` from the doubly linked list, we can simply link its previous node `prev` with its next node `next`.

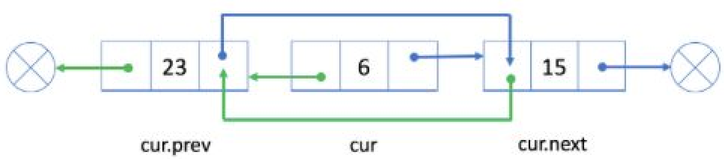
> Unlike the singly linked list, it is easy to get the previous node in constant time with the "prev" field.

Since we no longer need to traverse the linked list to get the previous node, both the time and space complexity are `O(1)`.

An Example

Our goal is to delete the node 6 from the doubly linked list.

So we link its previous node 23 and its next node 15:



Node 6 is not in our doubly linked list now.

> What if we want to delete `the first node` or `the last node`?

```python
def deleteAtIndex(self, index):
    """
    Delete the index-th node in the linked list, if the index is valid.
    :type index: int
    :rtype: void
    """
    if index < 0 or index >= self.size:
        return

    curr = self.head
    if index == 0:
        self.head = curr.next
    else:
        for i in range(index - 1):
            curr = curr.next
        curr.next = curr.next.next

    self.size -= 1
```