Date __/__/__
Page

Problem statement: Segregate even and odd nodes in a linked list.

What is a linked list?

A linked list is a data structure used for storing lists. There are 2 ways of storing lists in memory.

1) Arrays
2) Linked lists.

Array elements are stored in contigious memory locations but linked lists nodes can be scattered throughout the memory.

Linked list comprises of nodes and node consists of two main parts.

Data: Actual data that we store in memory.

Link: It contains the address of the next node in the list.

- To access the first node in the list we use a 'head' variable that stores the address of the first node.
- You can then access the other nodes using the "next" pointer present in every node.

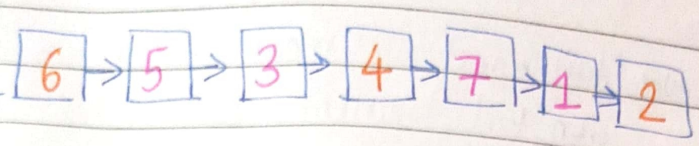
There are 3 types of linked lists.

1. Singly linked list: Each node points only to next node
2. Doubly linked list: Each node points to both next and previous nodes allowing bidirectional traversal.
3. Circular linked list: The last node points back to the first node creating a loop.

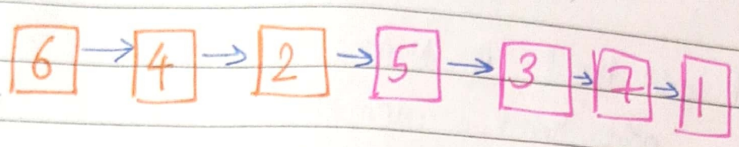
Date __/__/____
Page _____

To segregate even and odd nodes in a linked list.

Given linked list



modified linked list



Constraint: Don't create a new linked list

Python Code:

```
class ListNode:
    def __init__(self, data):
        self.data = data
        self.next = None

def segregate_even_odd(head):
    if head is None:
        return None

    even_head = None
    even_tail = None
    odd_head = None
    odd_tail = None

    current_node = head

    while current_node:
        if current_node.data % 2 == 0:
            if even_head is None:
```

Date __/__/__
Page

```
            even_head = current_node
            even_tail = current_node.
        else:
            even_tail.next = current_node
            even_tail = current_node
    else:
        if odd_head is None:
            odd_head = current_node
            odd_tail = current_node
        else
            odd_tail.next = current_node
            odd_tail = current_node.

        current_node = current_node.next.

    if even_head is None:
        return odd_head.

    even_tail.next = odd_head.

    if odd_tail:
        odd_tail.next = None

    return even_head

def create_linked_list(arr):
    if not arr
        return None
    head = ListNode(arr[0])
    current = head
    for i in range(1, len(arr)):
        current.next = ListNode(arr[i])
```

Date __/__/____
Page _____

```
    current = current.next
return head

def print_linked_list(head):
    current = head
    while (current):
        print(current.data, end="->")
        current = current.next
    print("None")

arr = [17, 15, 8, 12, 10, 5, 4, 1, 7, 6]
head = create_linked_list(arr)

segregate_head = segregate_even_odd(head)

print("Segregated linked list:")
print_linked_list(segregate_head)
```

Output:

Original linked list :

17 -> 15 -> 8 -> 12 -> 10 -> 5 -> 4 -> 1 -> 7 -> 6

Segregated linked list :=

8 -> 12 -> 10 -> 4 -> 6 -> 17 -> 15 -> 5 -> 1 -> 7

Time Complexity : O(n)

Space Complexity : O(1)

Date __/__/__
Page

Code Explanation.

segregateEvenOdd (head) takes one parameter called head which as we discussed stores the address of the first node.

If head is null it means that the linked list is empty so it returns null.

Some variable are initialized to null and current node is initialized to the head value which means it stores the address of the first node for now.

1st iteration : While currentnode is not null it checks if the data is even first, by checking if the remainder of the data is 0 or not upon division with 2. In our example 6%2=0. so it enters the loop.
If even head is null which it is it will set the even head and even tail as 6. It skips all the all other statements and the current node contains the address of 5 now.

2nd iteration:
current node is not null it contains the address of 5 so it enters the while loop. Since 5 is odd it enters the else loop oddhead is null so it enters the if block and now the oddhead as well as oddtail is set to 5. CurrentNode is 3 now

3rd iteration:
current node is not null it contains the address

Date __/__/____
Page _____

of 3 so it enters the while loop. Since 3 is odd it enters the else loop oddhead is not null it is 5 so it enters the else block and now the oddtail.next which means the 2nd node's link part will contain the address of the next odd node (data 3's address) odd tail will be 3 now. currentNode is 4 now.

The cycle repeats until currentNode becomes null in which case controls comes out of the while loop.

Now we will combine both the lists:

1. If even head is null it indicates that the linked list had no even nodes and hence it returns the oddhead which the first node of the linked list.
2. If even head is not null the last even element should contain the address of the 1st odd element so 2 will point to 5. This is how we join the two lists.
3. If oddtail has an element then it should point to null to end the linkedlist.
   Meaning | points to null.
4. In the end we return the evenhead which becomes the new first element's address and this becomes the new head.

Republic Gold

Date __/__/____
Page _____

Problem Statement: Convert min heap to max Heap.

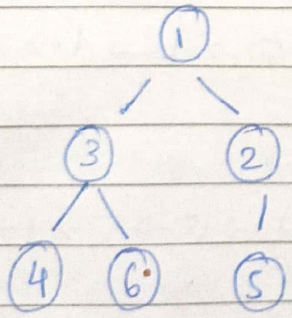
Heaps are tree-like data structures that satisfy the heap property. Heaps are like complete binary trees where all levels are filled except possibly the last level.
The nodes are filled will from left to right only and you cannot move to the next level until the previous level is full.

The structural property of a heap follows the almost complete binary tree (ACBT).

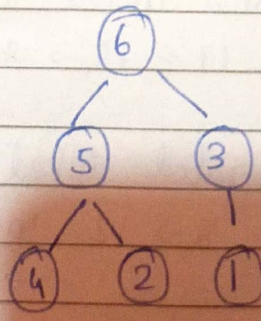
There are 2 types of heap:

→ Min heap: The smallest element is always at the root. At every level the nodes must be smaller than their child nodes.

→ Max heap: The largest element is at the root. At every level the nodes must be greater than their child nodes.



Min heap.



Max Heap.

Constructing Heap:

1. Insert key one by one.

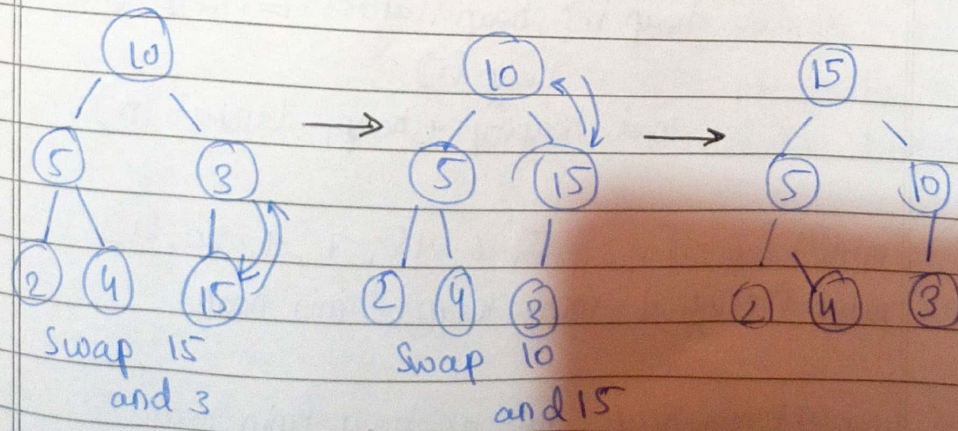
Heaps are often implemented using arrays.

Date __/__/__
Page

Set the first elements of the array as the root node and the 2nd element as child node
Compare both nodes and apply heap property (if you are constructing a max heap tree then see which node is bigger and swap accordingly)
Repeat this process from left to right until you reach the end of the array.
As you move to new levels check if the tree you created so far is satisfying the heap property or not and swap accordingly.
This method is a top to bottom approach.
The time complexity here o(nlogn) so we use the heapify approach.

2. Heapify method.

- Start with the non leaf nodes ie; start from 2nd last level as this is a bottom to top approach.
- Compare the nodes with its child nodes and apply the heap property (if you are constructing a max heap tree then see which node is greater in the child nodes and swap accordingly)
- Repeat this method for all subtrees recursively as you move upwards.



Republic Gold

Date __/__/__
Page

Python code:

```
import math

def convert_min_heap_to_max_heap(heap):
    n = len(heap)
    last_non_leaf_node_index = math.floor(n/2) - 1

    for i in range(last_non_leaf_node_index, -1, -1):
        max_heapify(heap, i, n)

def max_heapify(heap, i, n):
        largest = i
        left = 2*i + 1
        right = 2*i + 2

        if left < n and heap[left] > heap[largest]:
            largest = left

        if right < n and heap[right] > heap[largest]:
            largest = right

        if largest != i
        heap[i], heap[largest] = heap[largest],
            heap[i]
        max_heapify(heap, largest, n)

min_heap = [1, 3, 5, 6, 15, 7, 9, 20, 21]
print("original min-heap", min_heap)

convert_min_heap_to_max_heap(min_heap)
print("converted max-heap", min_heap)
```

Date __/__/____
Page _____

output:
original min heap:
[1, 3, 5, 6, 15, 7, 9, 20, 21]
converted max heap:
[21, 20, 9, 6, 15, 7, 5, 1, 3]

Time complexity : O(n)
Space complexity: O(1)

Code Explanation:

1. The function convert min heap to max heap takes a min heap represented as an array as input.
2. It takes the length of the heap and stores it in variable n. It also calculates the last Non leaf Node index which is necessary because we start the max heapify operation from there
3. It iterates the loop from the 2nd last level to index 0 which is the root node and applies max heapify operation to every node it comes across
4. In the function max Heapify we store the root index at every level in i, this is now assumed to be the largest. We calculate the left and right nodes of this root index and see if it first exists (left < n) and then check if the left and right nodes are greater than the parent node or not, if yes the variable largest gets updated with that value
5. If largest is not equal to I it definitely means that the root node was smaller than its child nodes and the swapping takes places
6. We then recursively call the max heapify function to check the subtree remains a max heap.

Republic Gold