RV College of Engineering®

Department of
Master of Computer Applications

MCA415SL
SKILL LAB

ASSIGNMENT

Linked List and Heap

Submitted by

RVCE24MCA017
KAJAL VIVEK SINGH CHAUHAN

RV COLLEGE OF ENGINEERING®, BENGALURU – 560059
(Autonomous Institution Affiliated to VTU, Belagavi)
Department of Master of Computer Applications

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 two ways of storing lists in memory, one is using arrays and the other method is using linked lists.

Then what is the difference between arrays and linked list?
Array elements are stored in contiguous memory locations but linked list nodes can be scattered throughout the memory.

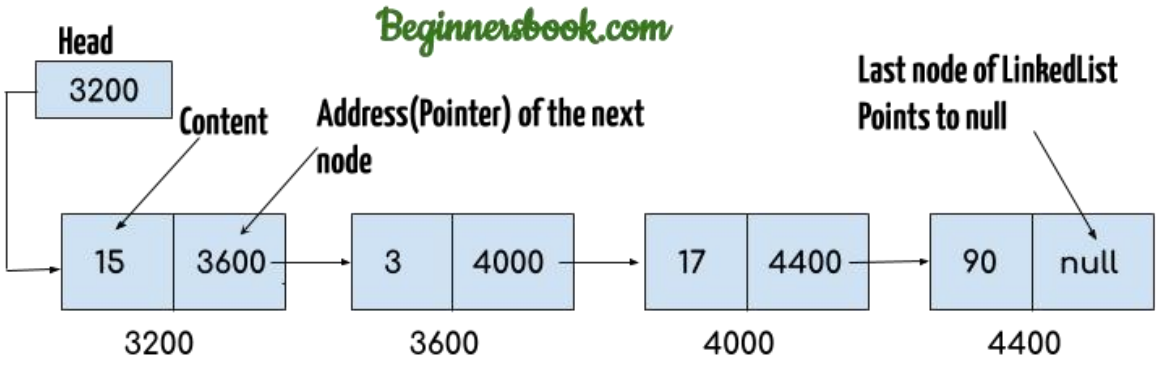
Linked lists comprises of nodes and a node consists of two main parts :

1. **Data :** The actual data we store in memory
2. **Link :** It contains the address of the next node in the list

How do we access the first node then?
We use a “head” variable to store the address of the first node. If the linked list is empty then the head pointer will have a value of “null”.

How to access a linked list?

- To access the first node you simply access the value of the head pointer.
- Once you have access to the first node you can then access its data and its "next" pointer which points to the next node in the list

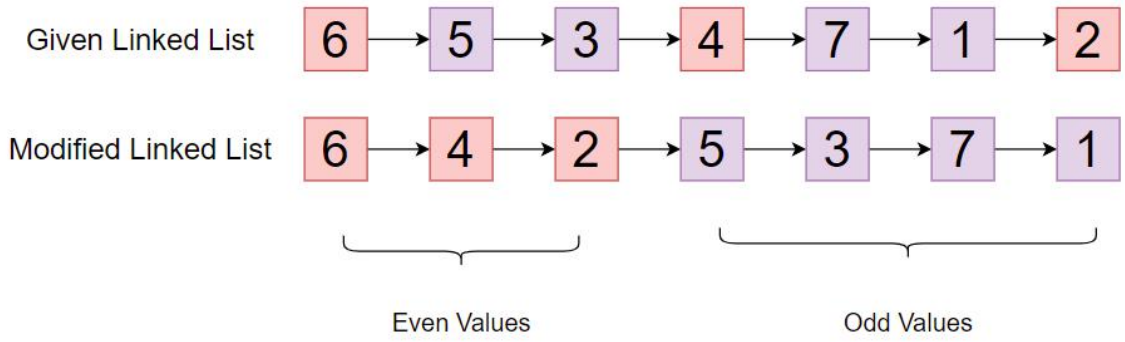


There are three types of linked lists:

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

Linked List is useful for dynamic memory allocation and efficiently add or delete elements especially in the middle of the list.

Now let’s try to understand the problem statement,



So basically given a linked list we have to segregate all the even values first and then the odd values

Constraint: Don’t create a new linked list instead rearrange the provided one.

PseudoCode:

```
FUNCTION segregateEvenOdd(head):
  # Input: head of a linked list
  # Output: head of the modified linked list with even nodes before odd nodes

  IF head is NULL:
    RETURN NULL

  evenHead = NULL
  evenTail = NULL
  oddHead = NULL
  oddTail = NULL

  currentNode = head

  WHILE currentNode is not NULL:
    IF currentNode.data MOD 2 equals 0: # Check if data is even
      IF evenHead is NULL:
        evenHead = currentNode
        evenTail = currentNode
      ELSE:
        evenTail.next = currentNode
        evenTail = currentNode
    ELSE: # Data is odd
      IF oddHead is NULL:
        oddHead = currentNode
        oddTail = currentNode
      ELSE:
        oddTail.next = currentNode
```

```
      oddTail = currentNode

   currentNode = currentNode.next

# Combine even and odd lists
IF evenHead is NULL: # No even nodes
   RETURN oddHead

evenTail.next = oddHead

IF oddTail is not NULL:
   oddTail.next = NULL # Terminate the odd list to avoid cycles

RETURN evenHead
```

PseudoCode Explanation:

segregatEvenOdd(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 variables are initialized to null, and currentnode 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 currentnode 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 it contains the address 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) oddtail will be 3 now. currentNode is 4 now.

The cycle repeats until currentNode becomes null in which case control 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 is 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 linked list.

Meaning 1 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.

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:
                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

# Helper function to create a linked list from a list
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])
        current = current.next
    return head

# Helper function to print a linked list
def print_linked_list(head):
    current = head
    while current:
        print(current.data, end=" -> ")
        current = current.next
    print("None")

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

print("Original Linked List:")
print_linked_list(head)

segregated_head = segregate_even_odd(head)

print("Segregated Linked List:")
print_linked_list(segregated_head)

arr2 = [1,3,5,7]
head2 = create_linked_list(arr2)
print("Original Linked List 2:")
print_linked_list(head2)
segregated_head2 = segregate_even_odd(head2)
print("Segregated Linked List 2:")
print_linked_list(segregated_head2)
```

Output:

```
Original Linked List: 17 -> 15 -> 8 -> 12 -> 10 -> 5 -> 4 -> 1 -> 7
-> 6 -> None
Segregated Linked List: 8 -> 12 -> 10 -> 4 -> 6 -> 17 -> 15 -> 5 ->
1 -> 7 -> None
Original Linked List 2: 1 -> 3 -> 5 -> 7 -> None
Segregated Linked List 2: 1 -> 3 -> 5 -> 7 -> None
```

Time Complexity:

This algorithm traverses the linked list once, so the time complexity is O(n).

Space Complexity:

This algorithm uses a constant amount of space (oddhead, evenhead, eventail etc) and does not use any other data structure to solve the problem so the space complexity is O(1)

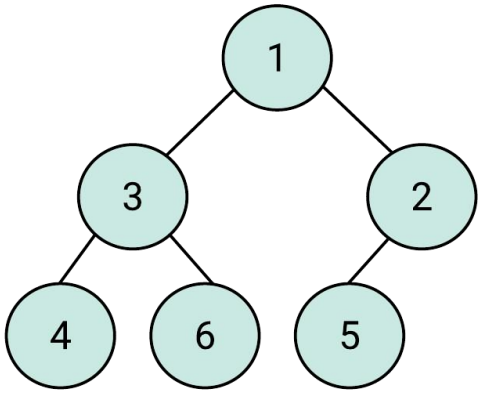
Problem Statement: Convert min heap to max heap

What is Heap?

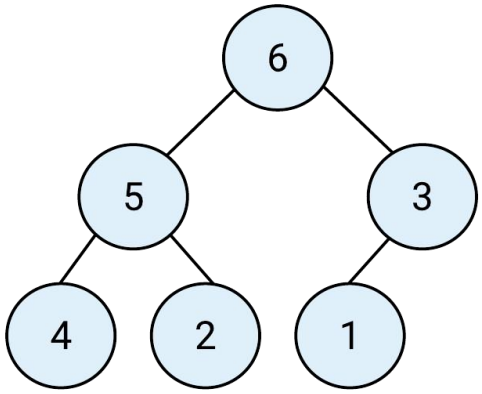
Heaps are tree-like data structures that satisfy the heap property. Heaps are like complete binary trees where all levels are filled with nodes except possibly the last level. The nodes are filled from left to right ONLY and you cannot move to the next level until the previous level is full. The structural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

There are two types of heap
1. **Min Heap** : The smallest element is always at the root. At every level the nodes must be smaller than their child nodes.

2. 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: (2 ways)

1. Insert key one by one

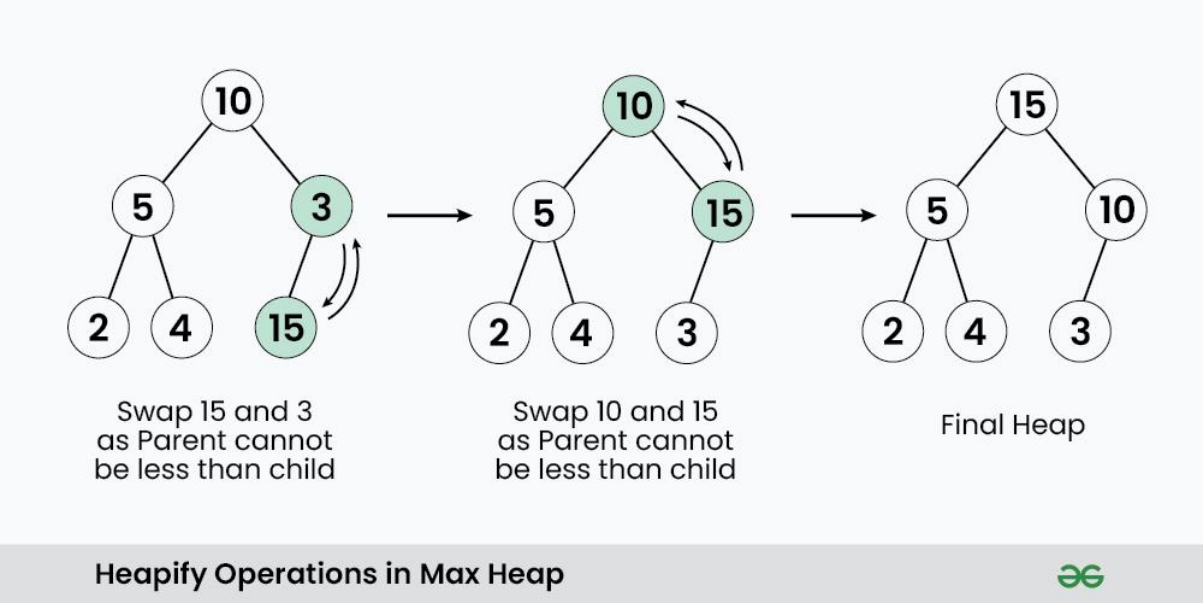
Heaps are often implemented using arrays.
Set the first element of the array as the root node and the 2nd element as child node.
Compare both nodes and apply heap property (if you are constructing a max heap tree then see which node is bigger and swap accordingly).
Repeat this process from left to right 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 is O(nlogn) so we use the heapify approach

2. Heapify Method

Start with the non leaf nodes I.e; start from the 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



Heapify Operations in Max Heap

Now let's try to understand the problem statement,

We are asked to convert min heap to max heap the steps that we can take to solve this problem are:

1. Start from the non leaf node which is located at index (n/2-1) where n is the number of elements is the heap
2. Iterate upwards till you read the root node
3. For each node perform the max-heapify operation and recursively apply it to affected sub trees as well.

PseudoCode:

```
function convertMinHeapToMaxHeap(heap):
  n = length(heap)
  lastNonLeafNodeIndex = floor(n / 2) - 1

  for i from lastNonLeafNodeIndex down to 0:
    maxHeapify(heap, i, n)

function maxHeapify(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:
    swap(heap[i], heap[largest])
    maxHeapify(heap, largest, n)

function swap(a, b):
  temp = a
  a = b
  b = temp
```

PseudoCode Explanation:

1. The function convertMinHeapToMaxHeap 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 lastNonLeafNodeIndex which is necessary because we start the maxheapify operation from there. We do not swap the leaf nodes.
3. It iterates the loop from the 2nd last level to index 0 which is the root node and applies maxheapify operation to every node it comes across
4. In the function MaxHeapify 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 smalled than its child nodes and the swapping takes place.
6. We then recursively call the maxheapify function to check the subtree remains a max-heap
7. Finally we have written the logic of swap operation.

Python Code:

```
import math

def convert_min_heap_to_max_heap(heap):
    """Converts a min-heap to a max-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):
    """Maintains the max-heap property."""
    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]  # Swap
        max_heapify(heap, largest, n)

# Example Usage
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)
```

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:

1. The convert_min_heap_to_max_heap function iterates through non leaf nodes so the time complexity of that function is O(n/2) where n=nodes
2. In worst case if if has to traverse the entire tree the max_heapify() = O(logn)
3. However total time complexity is O(n) because max heapify is performed generally on trees that have a small height

Space Complexity:

The space complexity is O(1) but if you consider the recursive stack space which is auxillary usually the it is O(log n)