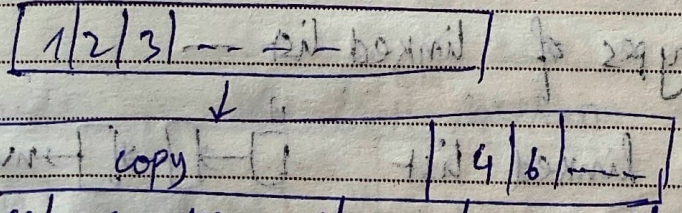
Linked List

Array list vs Linked list

insert: O(n) > O(1)

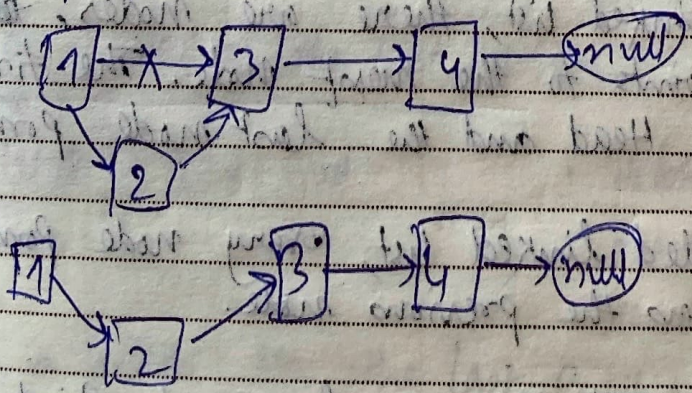
search: O(1) < O(n)

✓ In Array list, elements always get added in the form of dynamic array:



so, to insert a new element, we shall get a time complexity of O(n).

✓ In linked list, the memory is allocated in a ~~non-contiguous~~ non-contiguous manner. So, to add any element is a 2-3 step process



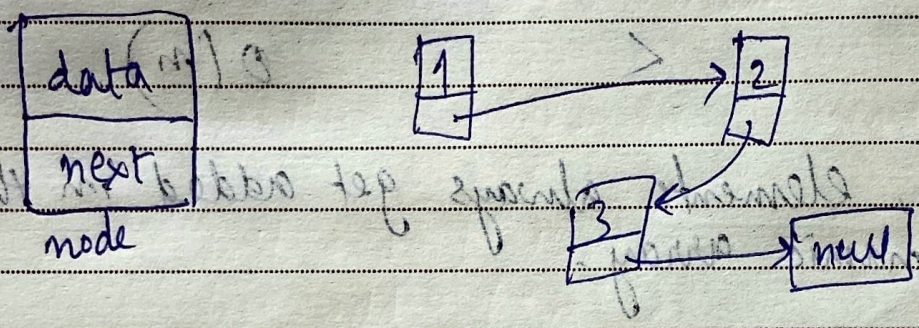
✓ In array list we can directly search an element by it's index. So, O(1).

✓ In linked list, we don't know where the element is. So, we need to search from the beginning. So, O(n).

NOTES

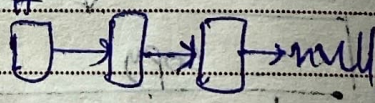
Properties of linked list

(i) variable size
(ii) Non-contiguous memory
(iii) Insert in O(1).
(iv) Search in O(n).



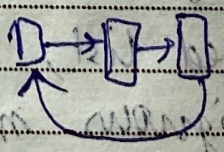
There are 3 types of linked list —

1) Singular linked list



2) Double-ended linked list

3) Circular linked list



✓ In singular linked list, there are nodes, and every node points to the next node. The first node is called the Head and the last node points to null.

✓ In Double-ended linked list, every node points to the next as well as the previous node.

✓ In circular linked list, the last node points to the first node and there exists no null.

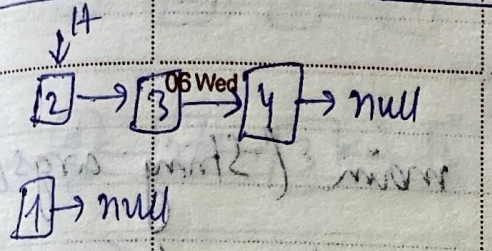
Action Plan

JUL '22

Singular Linked List

01 Fri 02 Sat 03 Sun 04 Mon

Add a node in the first position

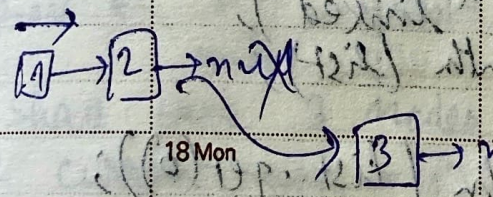


- the new node points to null initially.
- Link the new node to the previous head.
- Point the head to the new node.

05 Tue 06 Wed 07 Thu 08 Fri

09 Sat 10 Sun 11 Mon 12 Tue

Add a node in the last position

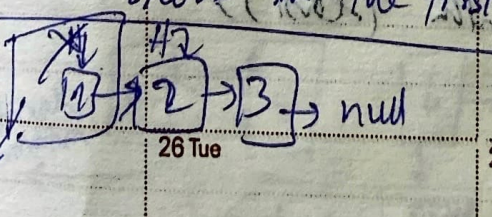


- Traverse from the 1st node and find the last node.
- Point the last node to the new node.
- next of new node already points to null.

13 Wed 14 Thu 15 Fri 16 Sat

17 Sun 18 Mon 19 Tue 20 Wed

Delete a node (in the first position)

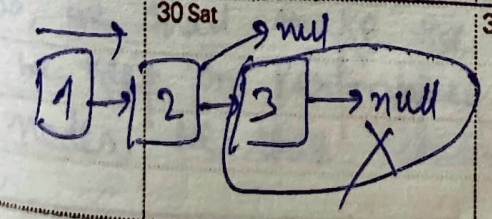


- delete head from the first position by making the second element as head.

21 Thu 22 Fri 23 Sat 24 Sun

25 Mon 26 Tue 27 Wed 28 Thu

delete a node in the last position



- Traverse from the first element. when 2nd last node is reached, point it to null. So, last node gets deleted.

29 Fri 30 Sat 31 Sun

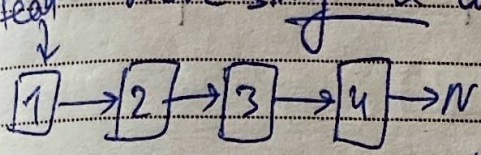
Linked list class implementation (collection framework)

```
import java.util.*;
class LL {
    public static void main (String args[])
    {
    LinkedList <String> list = new LinkedList <String>();
    list.add ("is");
    list.add ("a");
    list.addLast ("list");
    list.addFirst ("this");
    list.add (3, "linked");
    System.out.println (list);

    System.out.println (list.get(0));
    System.out.println (list.size());
    list.remove(3);
    list.removeFirst();
    list.removeLast();
    System.out.println (last);
    }
}
```

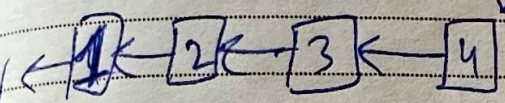
2022 AUGUST

July
27th Wk • 183-182
02
Saturday

Reversing a linked list



: Linked list

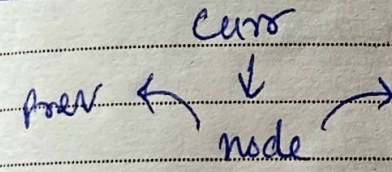


: Reversed linked list

note :- i) no extra memory to be consumed.

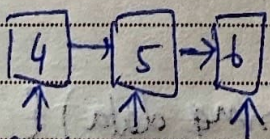
ii) Space complexity : O(1) shall not be changed.

Algorithm

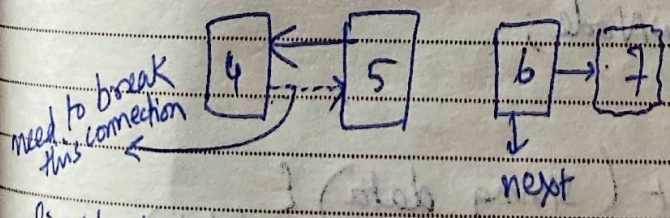


• we shall take 3 nodes set every time. Then, we shall make current. next = prev.

E.g.



Now, Curr. next = prev.



Similarly, next 3 nodes set and so on...

• Also, we shall make the next of first node as null.

If we keep on performing this action repeatedly for all nodes, we shall achieve a reverse linked list.

03 July
Sunday
27th Wk • 184-181

2022 JULY

Scratch Implementation of a linked list

```
class LL {
    Node head;
    private int size;

    LL() {
        size = 0;
    }

    Public class Node {
        String data;
        Node next;
        Node (String data) {
            this.data = data;
            this.next = null;
            size++;
        }
    }

    Public void addFirst (String data) {
        Node newNode = new Node (data);
        newNode.next = head;
        head = newNode;
    }

    Public void addLast (String data) {
        Node newNode = new Node (data);

        if (head == null) {
            head = newNode;
            return;
        }
```

```
Node lastNode = head;

while (lastNode.next != null) {
    lastNode = lastNode.next;
}
lastNode.next = newNode;
}

Public void printList() {
    Node currNode = head;
    while (currNode != null) {
        System.out.print (currNode.data + " → ");
        currNode = currNode.next;
    }
    System.out.println ("null");
}

Public void removeFirst () {
    if (head == null) {
        System.out.println ("Empty list, nothing to delete");
        return;
    }
    head = this.head.next;
    size--;
}
Public void removeLast () {
    if (head == null) {
        System.out.println ("Empty List, nothing to delete");
        return;
    }
```

```
        size --;
8       if (head.next == null) {
            head = null;
9           return;
        }
10      Node currNode = head;
        Node lastNode = head.next;
11
        while (lastNode.next != null)
        {
12          currNode = currNode.next;
            lastNode = lastNode.next;
1       }
            currNode.next = null;
2       }

3   Public  int getSize() {
            return size;
4   }

5   Public static void main (String args[]) {
        LL list = new LL();
6       list.addLast("is");
        list.addLast("a");
        list.addLast("list");
        list.PrintList();
        list.addFirst("this");
        list.PrintList();
        System.out.println(list.getSize());
        list.removeFirst();
        list.removeLast();
        list.PrintList();
    }
```

How to insert in the middle of a linked list ?

Scratch implementation

```
import java.util.*;
class LL{
public static void main(String args[]){
LinkedList<String> list = new LinkedList<String>();
list.addFirst("Indranil");
list.addFirst("name");
list.addFirst("my");
System.out.println(list);
list.add(2,"is");
System.out.println(list);
}
}
public void addInMiddle(int index, String data)
{
if(index > size || index < 0){
System.out.println("Invalid Index value");
return;
}
size++;
Node newNode = new Node(data);
if(head == null || index == 0){
newNode.next = head;
head = newNode;
return;
}
Node currNode = head;
for(int i=1; i<size; i++)
{
```

07 Thursday July 28th Wk • 188-177

2022

```
        if (i == index)
        {
8           Node nextNode = currNode.next;
9           currNode.next = newNode;
            newNode.next = nextNode;
10          break;
        }
11      currNode = currNode.next;
    }
12  }
```

** Reversing a Linked List **

1

Q) Reverse a linked list without extra spaces

2

E.g. old list

3 1 → 2 → 3 → 4 → null

new list

4 4 → 3 → 2 → 1 → null

5 ✓ Iterative method

Time complexity → O(n)

6 Space complexity → O(1)

→ Public void reverseList ()

```
{
    if (head == null || head.next == null)
    {
        return;
    }
```

```
Node prevNode = head;
Node currNode = head.next;

while (currNode != null) {

Node nextNode = currNode.next;
currNode.next = prevNode;
prevNode = currNode;
currNode = nextNode;
}
head.next = null;
head = prevNode;
}
```

Recursive method

Time Complexity = O(n)
Space Complexity = O(1)

```
Public Node reverseListRecursive(Node head){

// empty node // last node or only one node
if(head == null || head.next = null)
{
    return head;
}
Node newHead = reverseListRecursive(head.next);

head.next.next = head;
head.next = null;
return newHead;
}
```

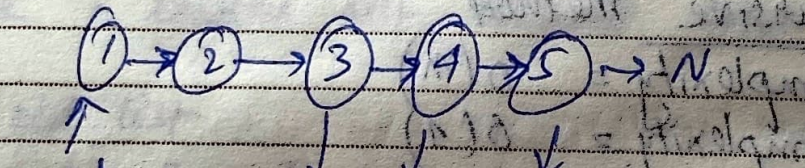
Collections Method

Time Complexity = O(n)
Space Complexity = O(1)

```
LinkedList<Integer> list2 = new LinkedList<>();
list2.add(1);
list2.add(2);
Collections.reversed(list2);
```

Most Important Linked List Question

Q1 Find the n^{th} node from last + delete n^{th} node.



Algorithm :-

① Find the size of the list.
here n=5.

② then apply a simple math logic.
nth node from last = (size − n + 1) from first

E.g., n=2 from last
(size − n + 1) = (5 − 2 + 1) = 4th node from 1st.

③ we will delete the nth node from last.
prev.next = prev.next.next;

Distance from start (dfs) => (size − n) => 5 − 2 = 3. → (prev. node)

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

Code :-

```
public ListNode removeNthFromEnd (ListNode head, int n)
{
    if (head.next == null)
    {
        return null;
    }
    int size = 0;
    ListNode temp = head;
    while (temp != null) {
            temp = temp.next;
            size++;
    }
// removing sizeth node from last, i.e head
    if (n == size) {
        return head.next;
    }
// find previous node
    int ptf = size - n; // position to find
    ListNode prev = head; // previous node
    int op = 1; // current position

    while (op != ptf) {
        prev = prev.next;
        op++;
    }
    prev.next = prev.next.next;
    return head;
}
```

Q2) check if the linked list is a palindrome or not.

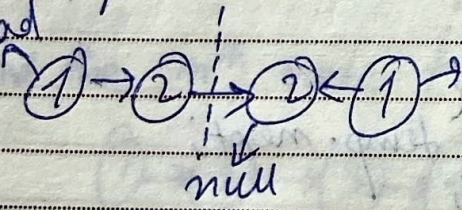
E.g. 1 → 2 → 1 → N

1 → 2 → 2 → 1 → N

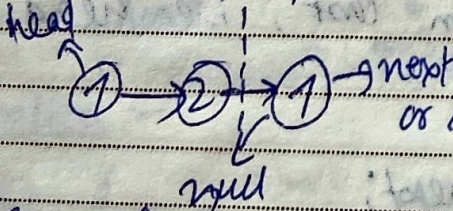
note :- no extra space can be utilized.

Algorithm

① Divide the linked list in half (so, now 2 LL)
② Reverse the second half of the linked list
③ compare both the linked list for palindrome

E.g 

If 1st half = 2nd half, Palindrome.



✓ Start from both heads and then traverse by one element each, until you reach null.

② For reversing one half of the linked list —
✓ make the middle node as the new Head
✓ Reverse the half

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

2022 AUGUST

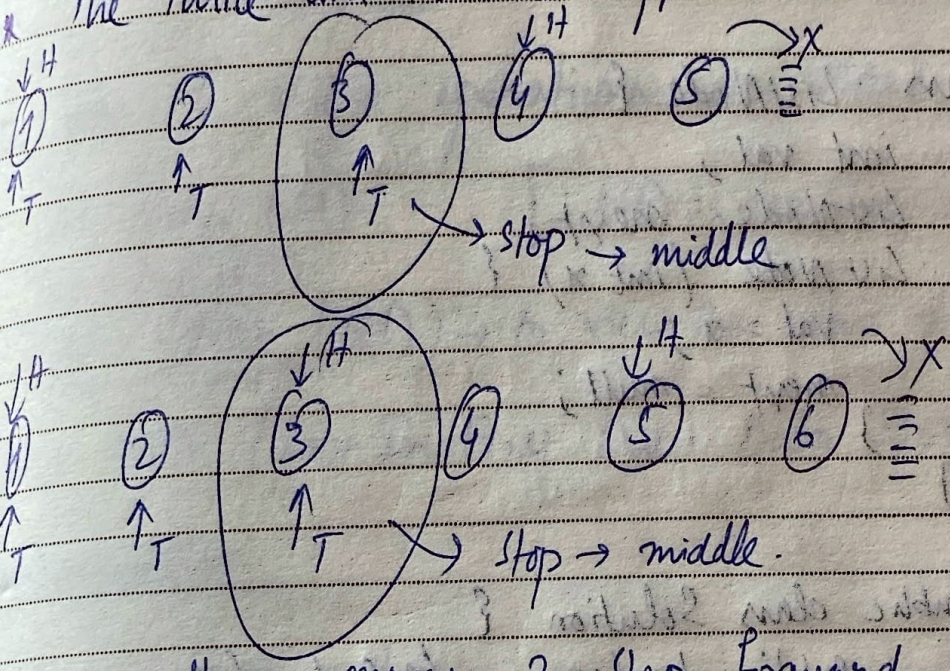
July
29th Wk • 193-172
12 Tuesday

Code :-

```
Public class ListNode {
    int val;
    ListNode next;
    ListNode () {
    }
    ListNode (int val) {
        this.val = val;
    }
    ListNode (int val, ListNode next) {
        this.val = val;
        this.next = next;
    }
}

class Solution {
    Public ListNode getMidde (ListNode head)
    {                                    // Turtle & Hare approach
        ListNode fast = head;            // Turtle = slow, fast = Hare
        ListNode slow = head;

        while (fast.next != null && fast.next.next != null){

            fast = fast.next.next;
            slow = slow.next;
        }
        return slow;
    }

    Public ListNode reverse (ListNode head) {
        ListNode prev = null;
        ListNode curr = head;
```

```
while (curr != null){
                    //interchange the pointers
        ListNode next = curr.next;
        curr.next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}

public boolean isPalindrome (ListNode head) {

    if (head == null || head.next == null) {
        return true;
    }

    ListNode firstHalfEnd = getMiddle (head);
    ListNode secondHalfStart = Reverse (firstHalfEnd.next);
    ListNode firstHalfStart = head;

    while (secondHalfStart != null) {
    if (secondHalfStart.val != firstHalfStart.val) {
        return false;
    }
        secondHalfStart = secondHalfStart.next;
        firstHalfStart = firstHalfStart.next;
    }
    return true;
}
}
```

The turtle and the hare Approach



✓ whenever Hare moves 2 step forward, Turtle moves 1 step forward.

✓ when Hare goes out of the range, Turtle stops. The stopping position of the Turtle is the middle position.

(23) Detecting a cycle in a linked list.

Approach :- Floyd's algorithm (Hare-Turtle Approach).

✓ we shall follow the same Hare-Turtle approach.

✓ Now, if it's cyclic linked list, then at some point the hare and the turtle is going to clash to one another.

If it clashes, then the linked list has a loop and if it doesn't, means the linked list has no loops.

Time complexity = $O(n)$

Space complexity = $O(1)$.

15 July
29th Wk • 196-169
Friday

2022 JULY

Code :-

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) {
        val = x;
        next = null;
    }
}

Public class Solution {
    public boolean hasCycle(ListNode head) {
        if (head == null) return false;
        ListNode slow = head;
        ListNode fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
            if (fast == slow) {
                return true;
            }
        }
        return false;
    }
}
```