**1. Problem Discussion :**

The problem states that: The function should be an iterative function and should return the middle of the linked list. make sure to not use size data member directly or indirectly (by calculating size via making a traversal) In the linked list of even size, consider the end of the first half as mid. Constraints: Size property should not be used directly or indirectly. Constant time, single traversal is expected. Iterative solution, (not recursion) is expected.

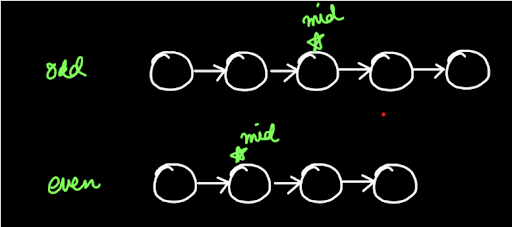
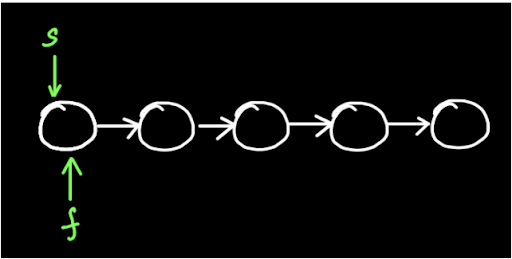
****

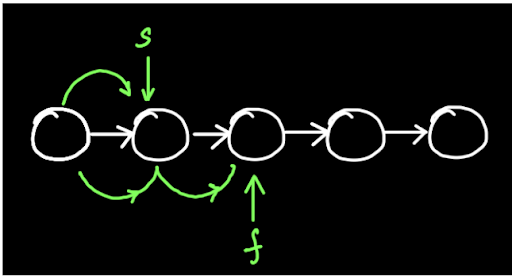
Figure 1 depicts the mid node for an odd and an even linked list. The strategy for this question is the Two Pointer Approach.

**2. Approach :**

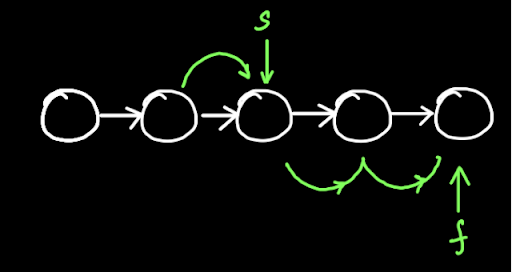
In this approach we initially put two pointers: s (slow) and f (fast) at the head of the given linked list as shown in figure 2.

****

For every 1 step of s pointer, the f pointer moves 2 steps (see figure 3).

****

This means that the speed of the 'f' pointer is twice that of the 's' pointer. So when the f pointer reaches the end, then the s pointer would be at the mid of the linked list (figure 4).

****

Hence, when 'f' points at the end node, then the node at the 's' pointer is the mid. Meditate on the fact that 'f' pointing at the last node can be described as f.next=null which means that the position of f pointer when there is no node after it. We have learnt the algorithm for this question, now let's code it quickly.

**3. CODE:**

ConsoleJava

import java.io.\*;

import java.util.\*;

public class Main {

public static class Node {

int data;

Node next;

}

public static class LinkedList {

Node head;

Node tail;

int size;

void addLast(int val) {

Node temp = new Node();

temp.data = val;

temp.next = null;

if (size == 0) {

head = tail = temp;

} else {

tail.next = temp;

tail = temp;

}

size++;

}

public int size() {

return size;

}

public void display() {

for (Node temp = head; temp != null; temp = temp.next) {

System.out.print(temp.data + " ");

}

System.out.println();

}

public void removeFirst() {

if (size == 0) {

System.out.println("List is empty");

} else if (size == 1) {

head = tail = null;

size = 0;

} else {

head = head.next;

size--;

}

}

public int getFirst() {

if (size == 0) {

System.out.println("List is empty");

return -1;

} else {

return head.data;

}

}

public int getLast() {

if (size == 0) {

System.out.println("List is empty");

return -1;

} else {

return tail.data;

}

}

public int getAt(int idx) {

if (size == 0) {

System.out.println("List is empty");

return -1;

} else if (idx < 0 || idx >= size) {

System.out.println("Invalid arguments");

return -1;

} else {

Node temp = head;

for (int i = 0; i < idx; i++) {

temp = temp.next;

}

return temp.data;

}

}

public void addFirst(int val) {

Node temp = new Node();

temp.data = val;

temp.next = head;

head = temp;

if (size == 0) {

tail = temp;

}

size++;

}

public void addAt(int idx, int val) {

if (idx < 0 || idx > size) {

System.out.println("Invalid arguments");

} else if (idx == 0) {

addFirst(val);

} else if (idx == size) {

addLast(val);

} else {

Node node = new Node();

node.data = val;

Node temp = head;

for (int i = 0; i < idx - 1; i++) {

temp = temp.next;

}

node.next = temp.next;

temp.next = node;

size++;

}

}

public void removeLast() {

if (size == 0) {

System.out.println("List is empty");

} else if (size == 1) {

head = tail = null;

size = 0;

} else {

Node temp = head;

for (int i = 0; i < size - 2; i++) {

temp = temp.next;

}

tail = temp;

tail.next = null;

size--;

}

}

public void removeAt(int idx) {

if (idx < 0 || idx >= size) {

System.out.println("Invalid arguments");

} else if (idx == 0) {

removeFirst();

} else if (idx == size - 1) {

removeLast();

} else {

Node temp = head;

for (int i = 0; i < idx - 1; i++) {

temp = temp.next;

}

temp.next = temp.next.next;

size--;

}

}

private Node getNodeAt(int idx) {

Node temp = head;

for (int i = 0; i < idx; i++) {

temp = temp.next;

}

return temp;

}

public void reverseDI() {

int li = 0;

int ri = size - 1;

while(li < ri){

Node left = getNodeAt(li);

Node right = getNodeAt(ri);

int temp = left.data;

left.data = right.data;

right.data = temp;

li++;

ri--;

}

}

public void reversePI(){

if(size <= 1){

return;

}

Node prev = null;

Node curr = head;

while(curr != null){

Node next = curr.next;

curr.next = prev;

prev = curr;

curr = next;

}

Node temp = head;

head = tail;

tail = temp;

}

public int kthFromLast(int k){

Node slow = head;

Node fast = head;

for(int i = 0; i < k; i++){

fast = fast.next;

}

while(fast != tail){

slow = slow.next;

fast = fast.next;

}

return slow.data;

}

public int mid(){

Node f = head;

Node s = head;

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

f = f.next.next;

s = s.next;

}

return s.data;

}

}

public static void main(String[] args) throws Exception {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

LinkedList list = new LinkedList();

String str = br.readLine();

while (str.equals("quit") == false) {

if (str.startsWith("addLast")) {

int val = Integer.parseInt(str.split(" ")[1]);

list.addLast(val);

} else if (str.startsWith("size")) {

System.out.println(list.size());

} else if (str.startsWith("display")) {

list.display();

} else if (str.startsWith("removeFirst")) {

list.removeFirst();

} else if (str.startsWith("getFirst")) {

int val = list.getFirst();

if (val != -1) {

System.out.println(val);

}

} else if (str.startsWith("getLast")) {

int val = list.getLast();

if (val != -1) {

System.out.println(val);

}

} else if (str.startsWith("getAt")) {

int idx = Integer.parseInt(str.split(" ")[1]);

int val = list.getAt(idx);

if (val != -1) {

System.out.println(val);

}

} else if (str.startsWith("addFirst")) {

int val = Integer.parseInt(str.split(" ")[1]);

list.addFirst(val);

} else if (str.startsWith("addAt")) {

int idx = Integer.parseInt(str.split(" ")[1]);

int val = Integer.parseInt(str.split(" ")[2]);

list.addAt(idx, val);

} else if (str.startsWith("removeLast")) {

list.removeLast();

} else if (str.startsWith("removeAt")) {

int idx = Integer.parseInt(str.split(" ")[1]);

list.removeAt(idx);

} else if(str.startsWith("reverseDI")){

list.reverseDI();

} else if(str.startsWith("reversePI")){

list.reversePI();

} else if(str.startsWith("kthFromEnd")){

int idx = Integer.parseInt(str.split(" ")[1]);

System.out.println(list.kthFromLast(idx));

} else if(str.startsWith("mid")){

System.out.println(list.mid());

}

str = br.readLine();

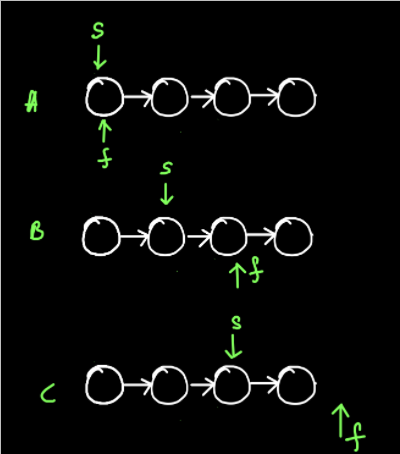
}

}

}

**4. CODE DISCUSSION**

In the while loop you must have noticed one more condition : f.next.next!=null. Why do you think it is for? Reader, it's for the even cases. Let's see how by first only checking the condition f.next!=null for a linked list of even size (see figure 5).

****

In figure 5(B) you can see that s pointer is at the desired mid location but since f.next is still not null, therefore the loop iterates again and s pointer has to move 1 more step at a position which is not the mid position. Hence, we check the condition f.next.next!=null , so that we move the f piece only when at least 2 nodes are present further.

**5. Analysis:**

Time Complexity: O(n) The time complexity is linear as a while loop is used in the program. Space Complexity: O(1) Since no extra space is used, therefore the space complexity is constant. You should also check out the solution video for this question.