Faculty of Engineering & Information Technology

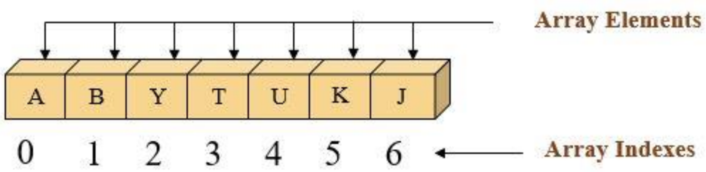
Data Structures & Algorithms

Fundamental Data Structures

Asst. Prof. Dr. Ahmed A.O. Tayeh

Arrays

- A sequenced collection of variables all of the same type
  - all integer, all float-point, etc
- Stores related data
  - students
  - university courses



- Has fixed size
  - gets a fixed size at initialisation time
  - should consider other data structures when dynamicity is needed
- Stored in successive memory locations
- *Array length*: maximum capacity
- *Array size*: actual number of stored elements

Array Operations

- **Traversal:** traverse all the elements one after another
- **Insertion:** add an element at a given position
- **Deletion:** delete an element at a given position
- **Searching:** search an element using a given index or value
- **Updating:** update an element at a given index
- **Sorting:** arrange elements in the array in a specific order
  - *try to create Java code to sort an array of integers*
- **Merging:** merge two arrays into one
  - *try to create Java code to merge two arrays*

Array Operations: Insertion & Deletion

- Insertion & deletion examples explained earlier were done in straightforward manner
  - insert or delete at a given position using the index
- Insertion & deletion can take more advance forms that require intensive operations – Imagine the following case
  - array of exam grades of size 10 - gradesArray [10]
  - teacher inputs the grades one by one
  - program stores the values starting with gradesArray[i=0]
    - we store the grades in higher positions only if the grade is higher than the grades before!
    - if the grade is lower than the one before, then put it before
  - in summary, what we want to achieve is a sorted array while inserting the elements

Array Operations: Insertion & Deletion…

- Program next grade is 75
  - current gradesArray status

70	77	79	84						

- To store the grade 75, it should be added at gradesArray[1]
  - shift all items (elements) “forward →” starting from position gradesArray[1]
  - shift 84 from gradesArray[3] to gradesArray[4]
  - shift 79 from gradesArray[2] to gradesArray[3]
  - shift 77 from gradesArray[1] to gradesArray[2]
  - imagine you have more stored items, and the array is larger than 10 elements!
- *Can you write this code, storing array items from user input & produce a sorted array?! Try Please!*

Array Operations: Insertion & Deletion…

- Imagine you want to delete 77 and keep the array in a good manner
  - current gradesArray status

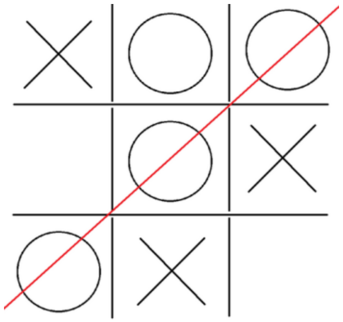
70	77	79	84						

- You need to remove 77 and then move all items that come after its position "← backward"
  - remove 77 from gradesArray[1]
  - move 79 from gradesArray[2] to gradesArray[1]
  - move 84 from gradesArray[3] to gradesArray[2]
- *Can you write this code, storing array items from user input and allowing user to delete elements and then keep the array in a good manner?! Try Please*

Multi-Dimensional Arrays

- A multi-dimensional array is an array of arrays
  - used in games (e.g., chess & Tic-tac-toe)
  - Mathematics (e.g., matrix calculation)

	Column 1	Column 2	Column 3	Column 4
Row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 3	a[2][0]	a[2][1]	a[2][2]	a[2][3]
Row 4	a[3][0]	a[3][1]	a[3][2]	a[3][3]



Multi-Dimensional Arrays…

- Java declaration;

data_type[1st dimension][2nd dimension][]..[Nth dimension] array_name = new data_type[size1][size2]….[sizeN];

  - data_type: type of data to be stored in the array
    - int, char (remember! homogenous values)
  - dimension: the dimension of the array; 1D, 2D, 3D, etc.
  - array_name: name of the array
  - size1, size2,…, sizeN: sizes of the dimensions respectively
- Two-dimensional array int[][] twoD_arr = new int[10][20];
- Three-dimensional array int[][][] threeD_arr = new int[10][20][30];
- Operations: traversing, insertion, deletion, searching, updating, sorting and merging

Multi-Dimensional Arrays…

```
public class TwoDimensionalArray {
    public static void main(String[] args) {
        int rows = 5;
        int columns = 5;
        int[][] array = new int[rows][columns];
        int value = 20;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                array[i][j] = value;
                value++;
            }
        }
        System.out.println("The 2D array is: ");
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                System.out.print(array[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Insertion - Updating

Traversing

Output:
The 2D array is:
20 21 22 23 24
25 26 27 28 29
30 31 32 33 34
35 36 37 38 39
40 41 42 43 44

Source: https://github.com/atayeh-israa-university/dataStructures-2023/blob/main/Theory%20-%20EITM2311/Multi-Dimensional-Arrays/TwoDimensionalArray.java

Array Advantages & Disadvantages

- **Advantages**
  - easy to use; store & access elements in contiguous memory blocks
  - random access; access by index. Find/retrieve elements by position (if you know it!)
  - performance; good if you use random access using element positions (search by index)
- **Disadvantages**
  - fixed size, once created cannot be changed, waste of memory –
  - lack of flexibility, you want more data, create another!
  - overhead; allocate certain amount of memory in advance
  - performance issues for inserting & deleting that requires shifting items

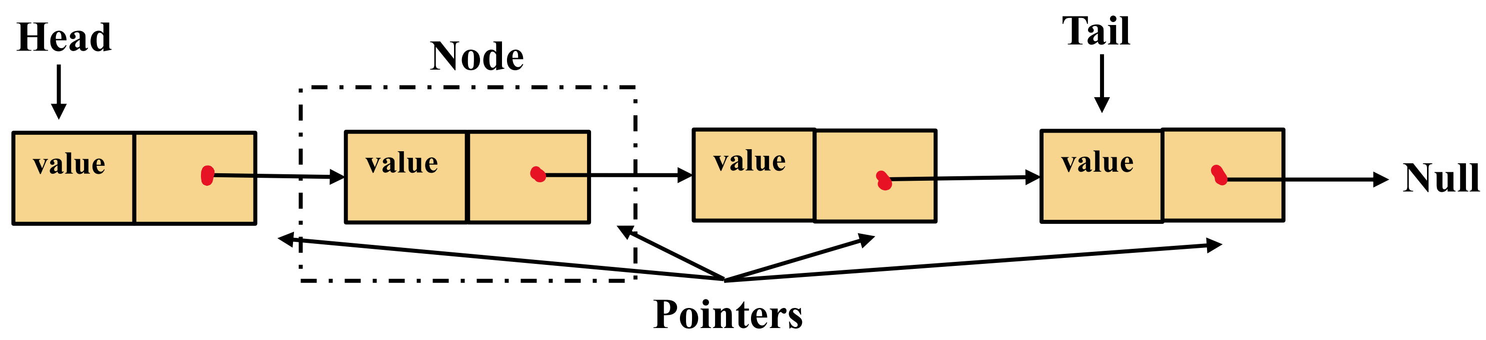
Arrays: Summary

- Know how and when to store your data in arrays
- Summarise advantages & disadvantages of arrays
- You should be able to write Java code to
  - define and create arrays of different primitive types
  - traverse arrays using for and enhanced for loops and search for elements
  - insert and update values
  - insert and delete & keep the array sorted
  - delete elements and keep the array with the same size
  - delete elements and shrink the array size
  - sort elements in arrays
  - merge more than one array in one single array
- Are you ready for a Quiz soon?

95% of theory & practicum exams are based on the examples you learn during the lectures & lab sessions

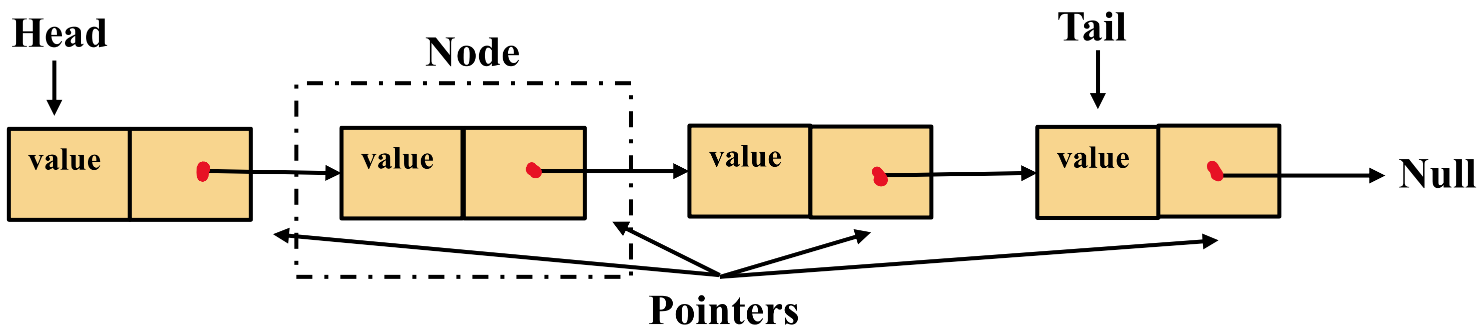
Singly Linked Lists

- Linear data structure storing many elements and dynamically extend/shrink its size
- Each element is called a node which contains two components; data and pointer “next”
  - “next” pointer points to the next node in the list
- First node is called head. The last called a tail and its “next” points to a null



Singly Linked Lists…

- Unlike arrays, nodes are not stored in a contiguous block of memory
- Each node holds the address of the next node in the list
- Accessing elements requires traversing the list from the head to the desired node
  - no direct access to a specific node
  - remember, arrays have indexes facilitating this



Singly Linked Lists…

```
//Class –  private? – representing the Node
class ListNode{
ListNode next;
int data;
public ListNode(){
next = null;
data = Integer.MIN_VALUE; //-2147483648, -2^31
}
public ListNode(int data){
next = null;
this.data = data;
}
```

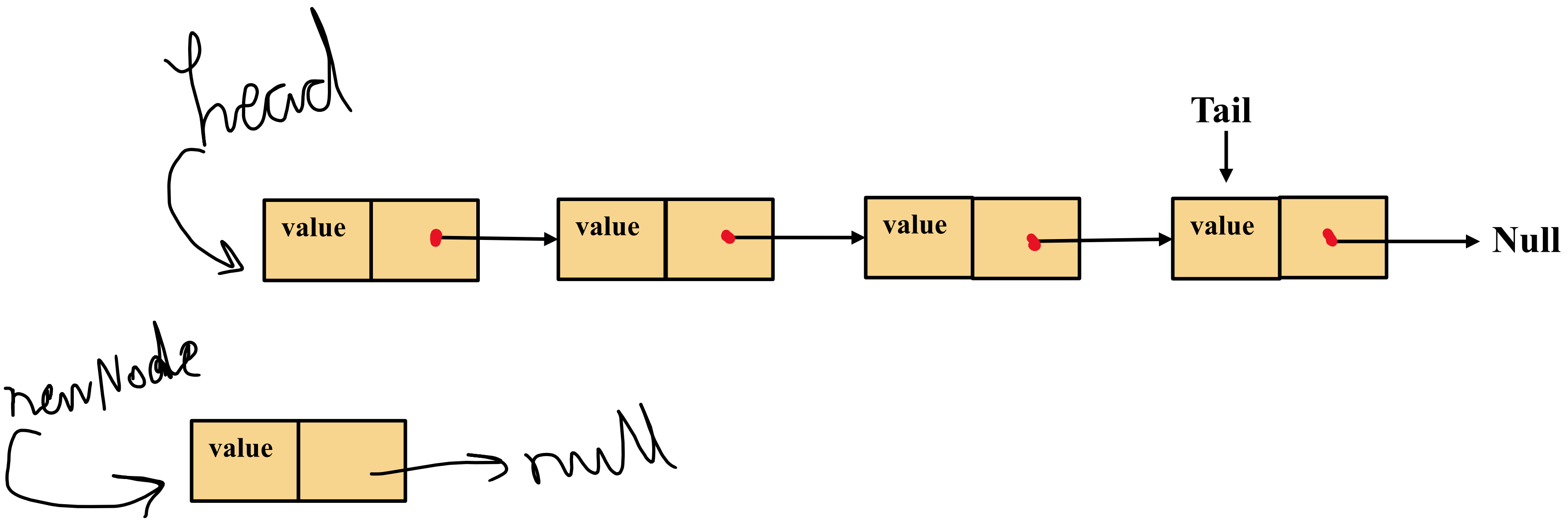
```
//class representing the list of Nodes
class LinkedList{
int length;
ListNode head;
ListNode tail;
public LinkedList(){
length = 0;
}
//methods for inserting (adding) nodes
void insertAtBegin(ListNode node){
//logic }
void insertAtEnd(ListNode node){
//logic }
void inset(int data, int position){
//traverse → calculate position → insert logic
}
//methods for removing nodes
….
```

Operations on Singly Linked Lists

- Insertion
  - at the head of the list
  - at the tail of the list
  - intermediate node
- Traversing & Searching
  - search for a value by its position or value
  - no explicit index assigned for a node in the list (must traverse!)
- Remove/Delete
  - first node
  - last node
  - intermediate node
- Size of the list or validate if list has elements or empty

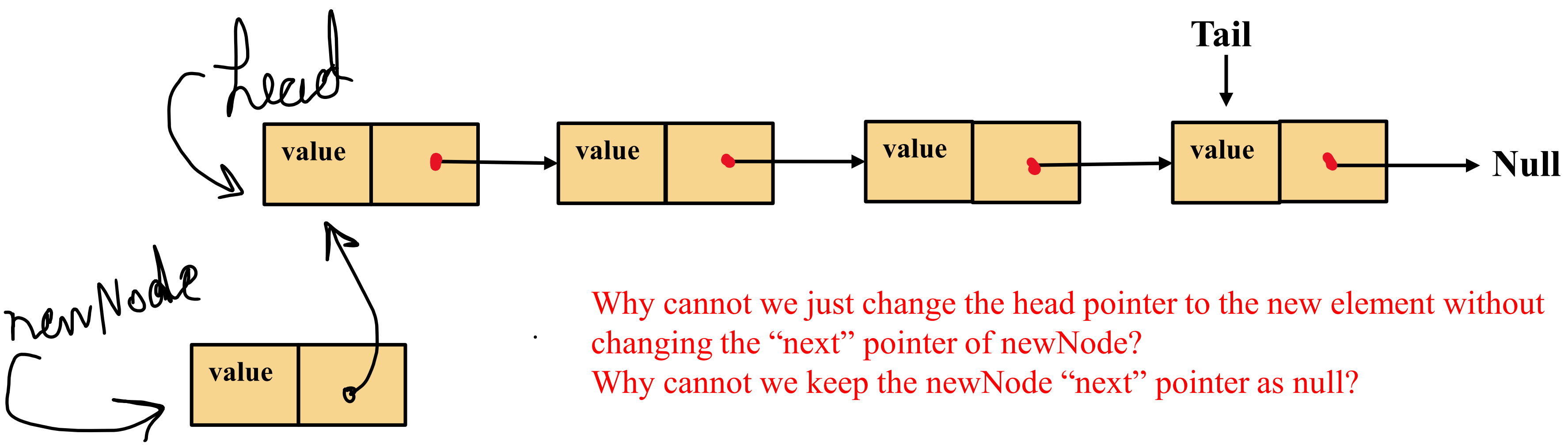
Singly Linked Lists: Insertion

- At the head of the list
  - allocate new node (newNode)
  - insert your data in the newNode



Singly Linked Lists: Insertion…

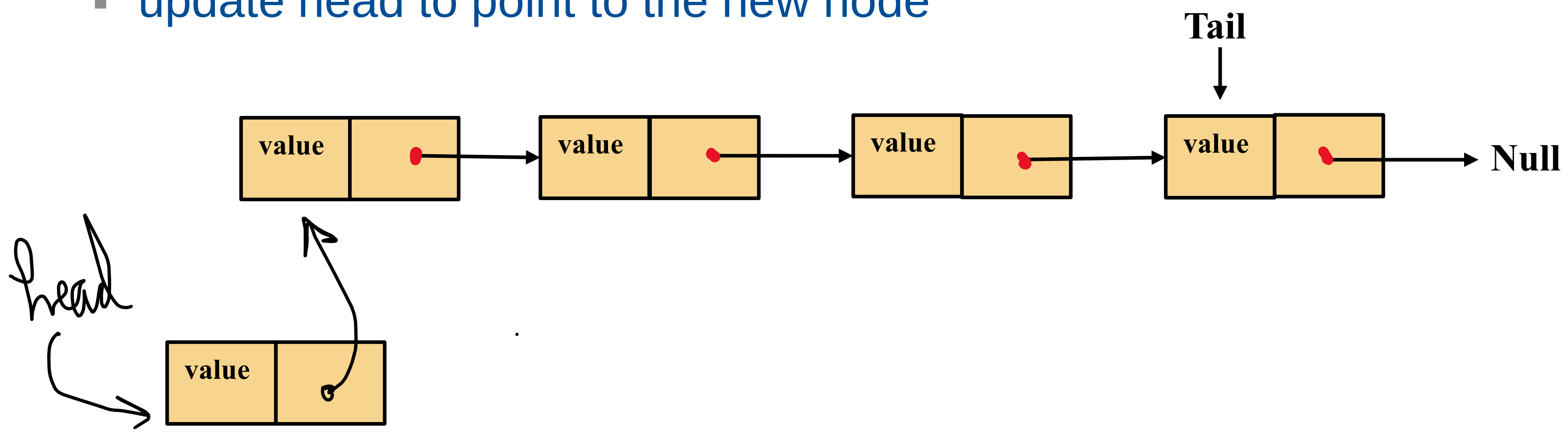
- At the head of the list (Cont.)
  - allocate new node (newNode)
  - insert your data in the newNode
  - NewNode now should also points to the old head (head still points to the same node)



Why cannot we just change the head pointer to the new element without changing the "next" pointer of newNode?
Why cannot we keep the newNode "next" pointer as null?

Singly Linked Lists: Insertion…

- At the head of the list (Cont.)
  - allocate new node (newNode)
  - insert your data in the newNode
  - NewNode now should also points to the old head (head still points to the same node)
  - update head to point to the new node



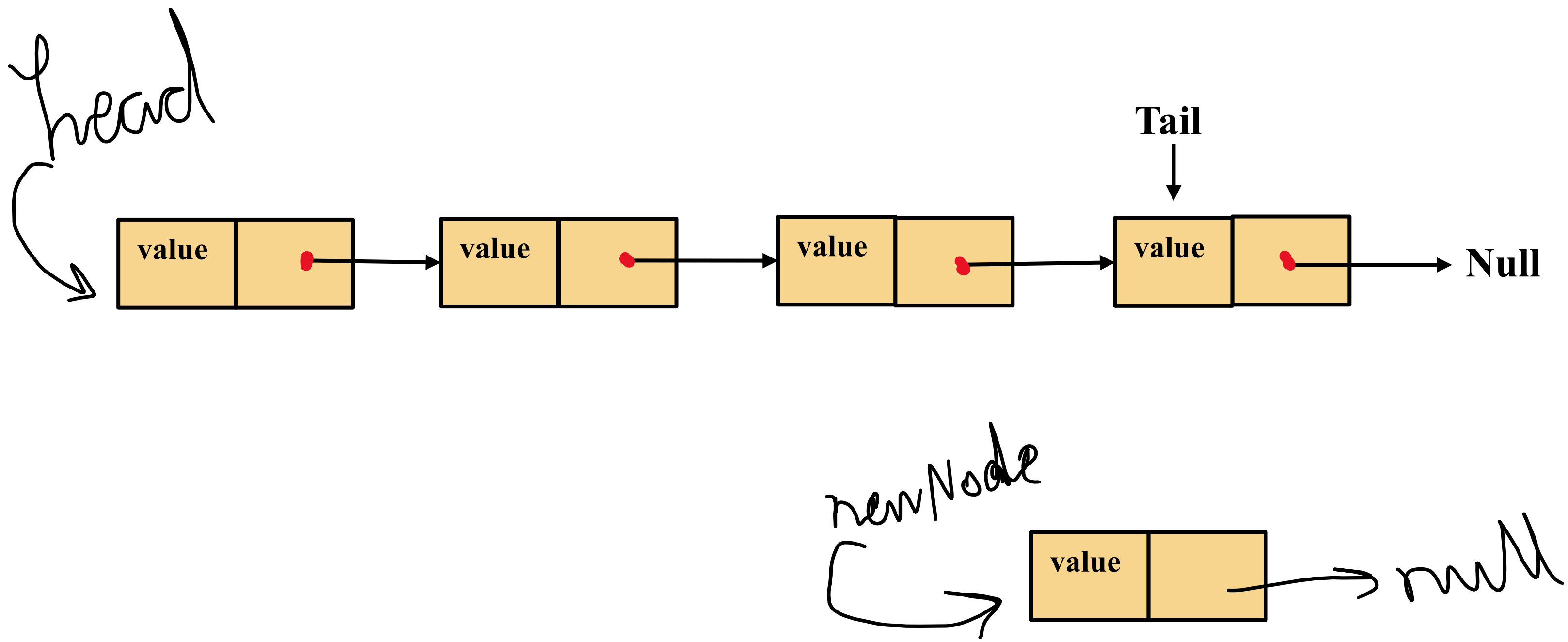
Singly Linked Lists: Insertion…

- At the head of the list (Cont.)

```
ListNode insertAtStart (int data)  {
     ListNode new_node = new ListNode (data); //initializing the new node
     new_node.next = head;  //new node points to the old head
     head = new_node; //head points to the new element
    return head;
  }
//Same behavior but different method parameters
ListNode insertAtStart(ListNode head, int data)     {
   ListNode newNode = new ListNode(data); //initializing the new node
   newNode.next = head; //new node points to the old head
   head = newNode; //head points to the new element
   return head;
    }
```

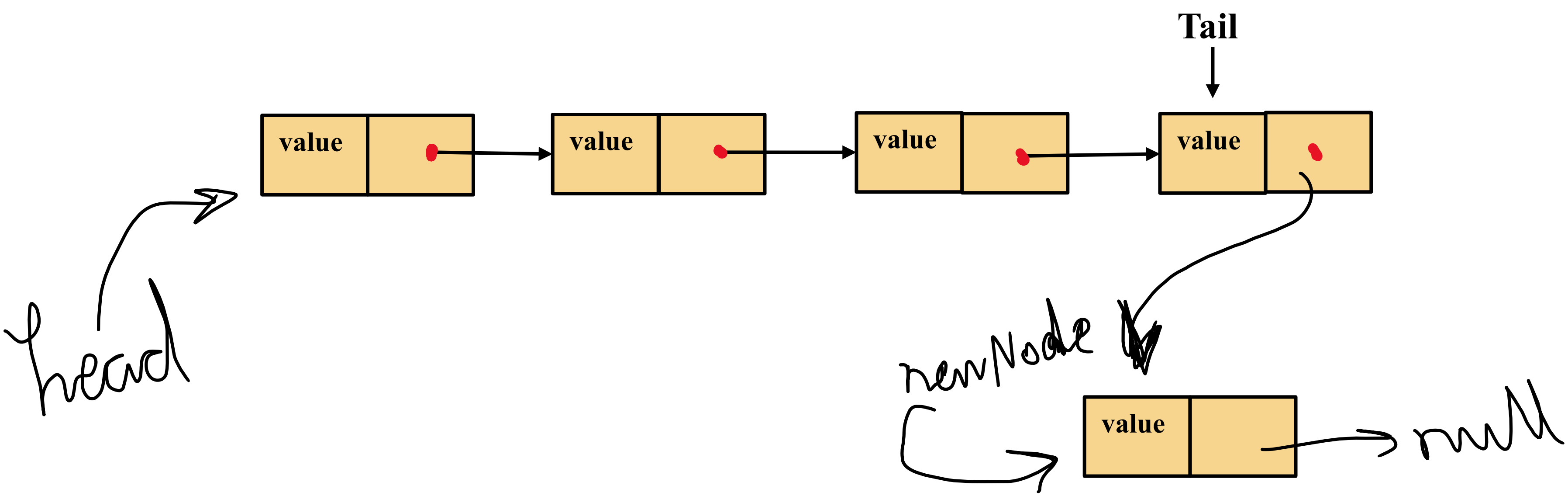
Singly Linked Lists: Insertion…

- At the tail of the list
  - allocate new node (newNode)
  - insert your data in the newNode
  - keep the newNode “next” pointer pointing to null. WHY?



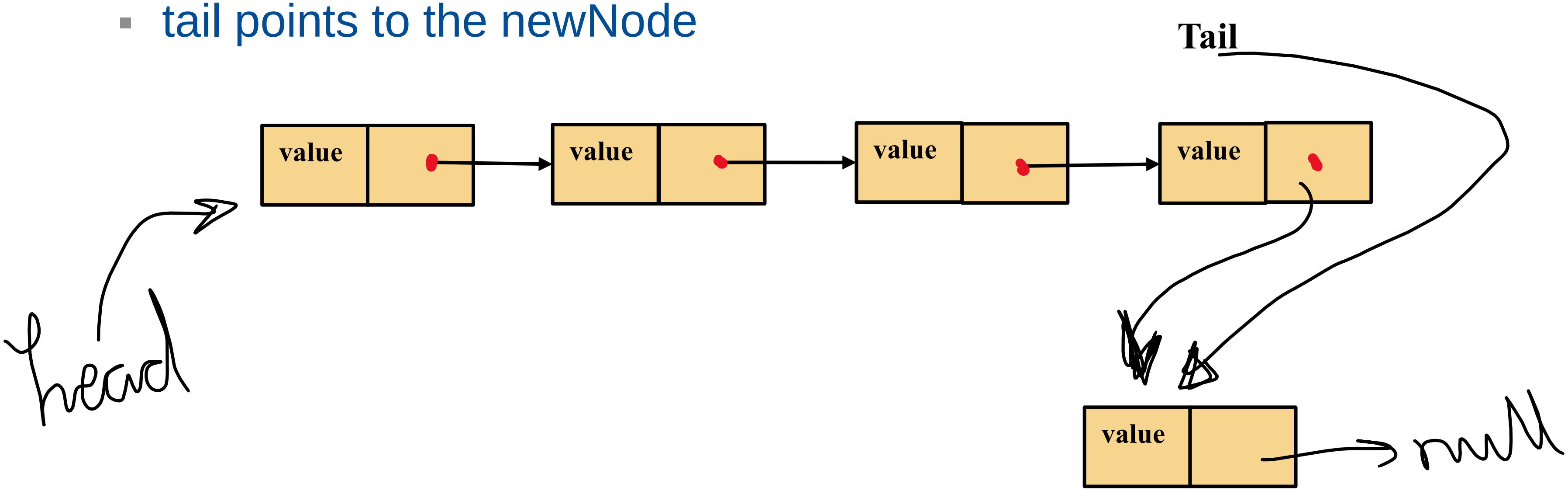
Singly Linked Lists: Insertion…

- At the tail of the list (Cont.)
  - allocate new node (newNode)
  - insert your data in the newNode
  - keep the newNode “next” pointer pointing to null
  - the last node in the list should then points to the newNode



Singly Linked Lists: Insertion…

- At the tail of the list (Cont.)
  - allocate new node (newNode)
  - insert your data in the newNode
  - keep the newNode “next” pointer pointing to null
  - the last node in the list should then points to the newNode
  - tail points to the newNode



Singly Linked Lists: Insertion…

- At the tail of the list (Cont.)

```
void insertAtEnd(ListNode node) {
//case 1: no nodes yet, then the newNode will be the first and last node
if (head == null) {
        head = node;
    } else {
        //you need two pointers
        ListNode p, q;
        for (p = head; (q = p.getNext()) != null; p = q) {
        }
        p.setNext(node);
    }
    length++;
}
```

Singly Linked Lists: Insertion…

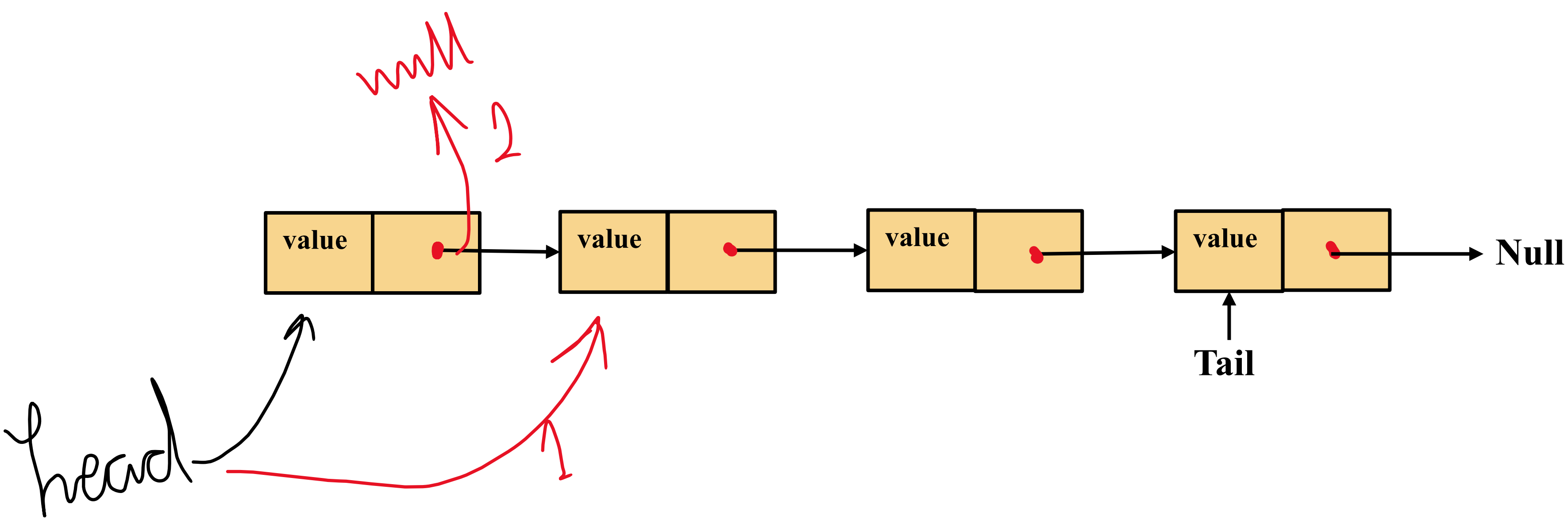
- Insert intermediate node at specific position
  - traverse and then change pointers – never break pointers

```
insert(int data, int position) {        // position should be checked not less than ZERO not larger than length
if (head == null) {     //list empty   -- add it first
head = new ListNode(data);     }
else if (position == 0) {     //adding it as first
ListNode temp = new ListNode(data);
            temp.next = head;    head = temp;       }
else {      // else find the correct position and insert
 ListNode temp = head;
//search-traverse and points to the position
for (int i = 1; i < position; i += 1) {     temp = temp.getNext();        }
            ListNode newNode = new ListNode(data);
            newNode.next = temp.next;
            temp.setNext(newNode);
           }
 length += 1; // the list is now one value longer
    }
```

Singly Linked Lists: Deletion

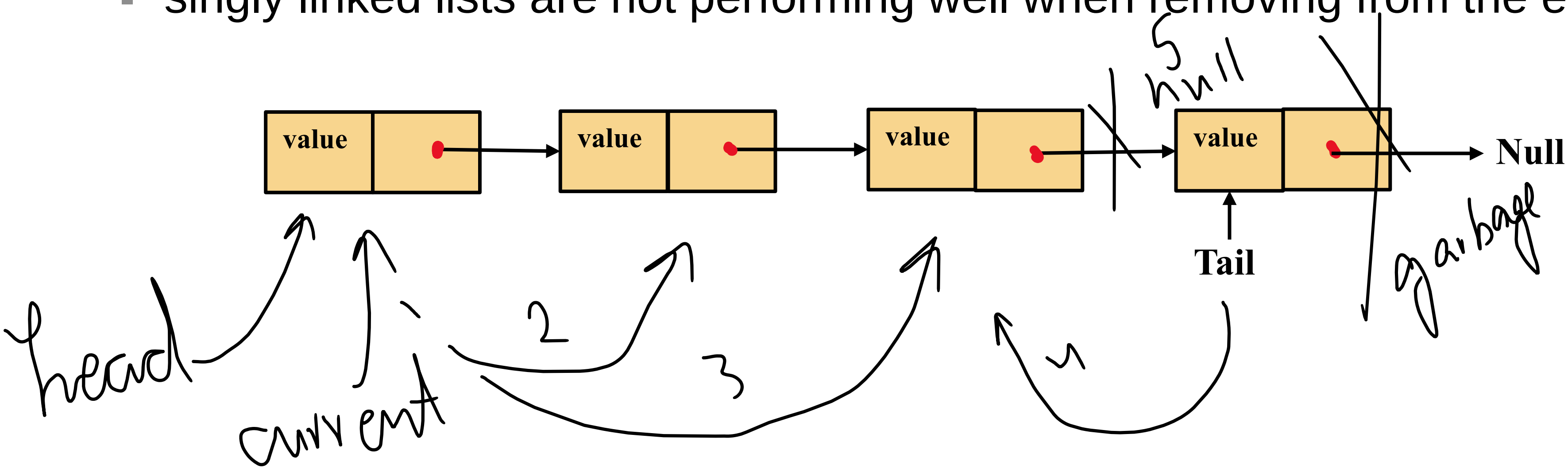
- Delete/remove the first element
  - head should point to the next node
  - set next of old head to null

```
Void removeFromBegin() {
        ListNode node = head;
        if (node != null) {
            head = node.getNext();
            node.setNext(null);
        }
}
```



Singly Linked Lists: Deletion…

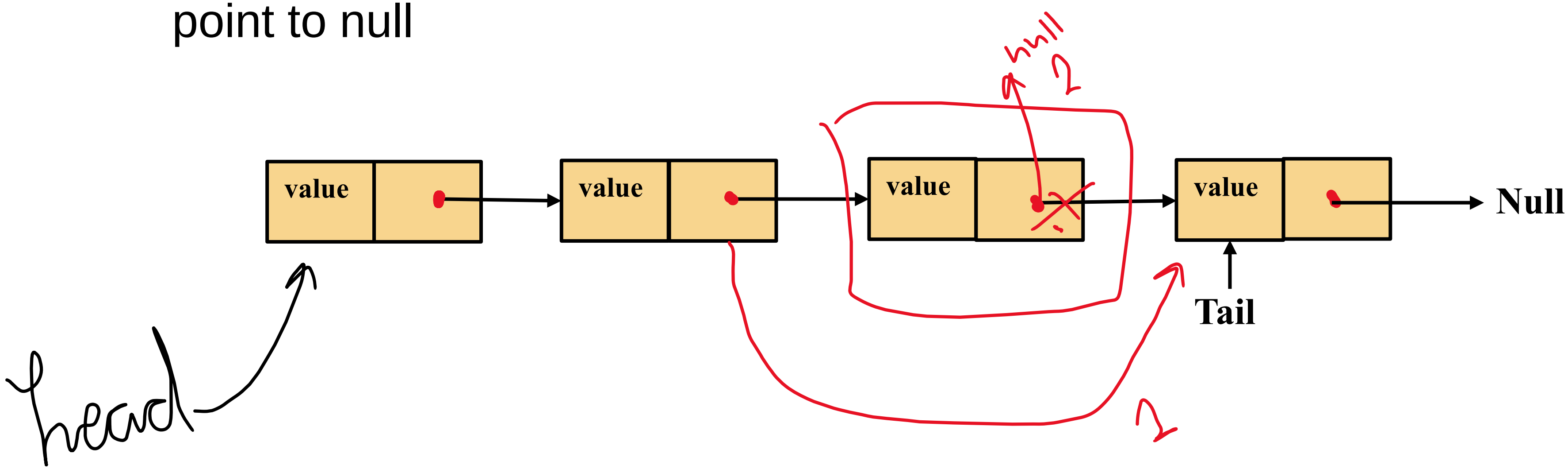
- Delete/remove the last element
  - a pointer “current” to traverse to the node before last – Why?
  - update the tail to point as the “current” pointer (i.e., the node before the last)
  - new tail “next” should be null
  - singly linked lists are not performing well when removing from the end



Singly Linked Lists: Deletion…

- Intermediate element
  - traverse to the element to be removed - <u>stop one node before – Why?</u>
  - next of the node before the element to removed should point to the node after the element to be removed
  - next of the node to be removed should point to null

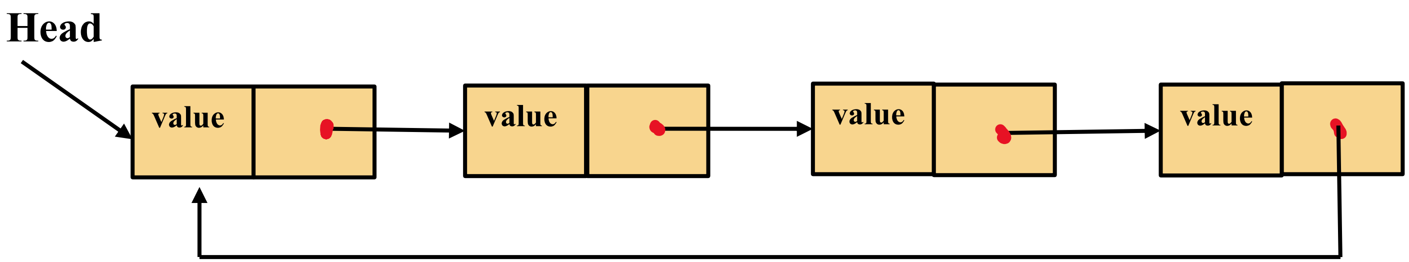
```
void removeIntermediate(Node node) {

  //Can you write the code?
}
```



Circular Singly Linked Lists

- Circular linked list is a linked list where all nodes are connected to form a circle.
  - the first and last nodes are connected to each other to form a circle
  - no null pointers at the end
  - you can traverse the list from any node and reach the node where you start from

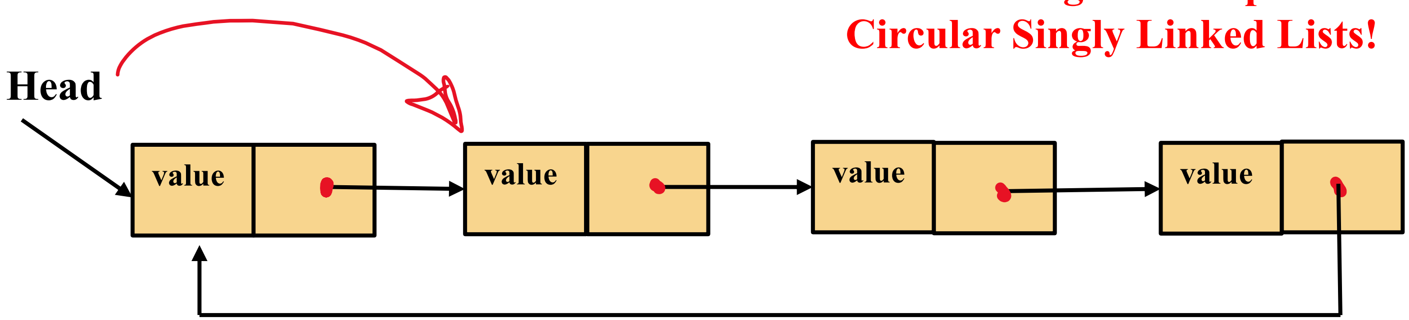
We will not go into implementation details of Circular Singly Linked Lists



Circular Singly Linked Lists…

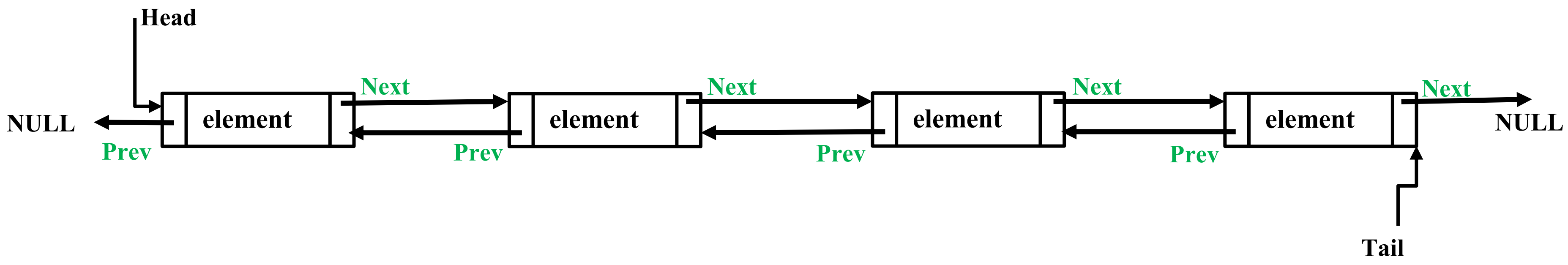
- Circular linked list is used in CPU round-robin process scheduling
  - allocate each task an equal share of CPU time
  - when task's allocated CPU time expires, the task is put at the end of the queue
  - take the next in queue
  - sometimes it is called circular queue

We will not go into implementation details of Circular Singly Linked Lists!



Doubly Linked Lists

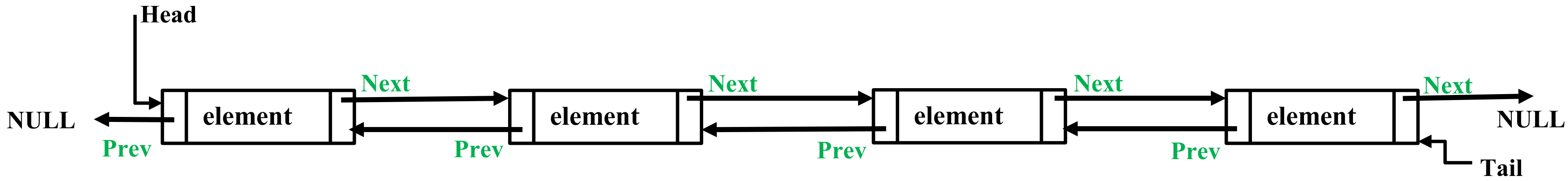
- Doubly linked list is a data structure that has reference to both the previous and next nodes.
  - has also Head (Header) and Tail (Trailer)
- Simple to traverse, insert and delete the nodes in both directions in a list
  - can be traversed in forward and backward directions



Doubly Linked Lists…

```
public class Node {
  int data;
  Node prev;
  Node next;
 public Node(int data)
 {
  this.data = data;
  this.prev = null;
  this.next = null;
 }
}
```

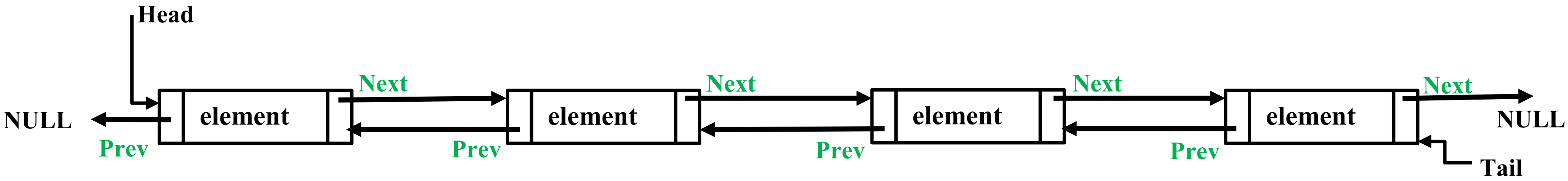
```
public class DoublyLinkedList {
   Node head;
   Node tail;

public DoublyLinkedList()
 {
  this.head = null;
  this.tail = null;
  }
}
```



Doubly Linked Lists: Traversing

```
// Traversing from head to the end of the list
public void traverseForward()
{
   Node current = head;
   while (current != null) {

   System.out.print(current.data + " ");
   current = current.next;
 }
}
```

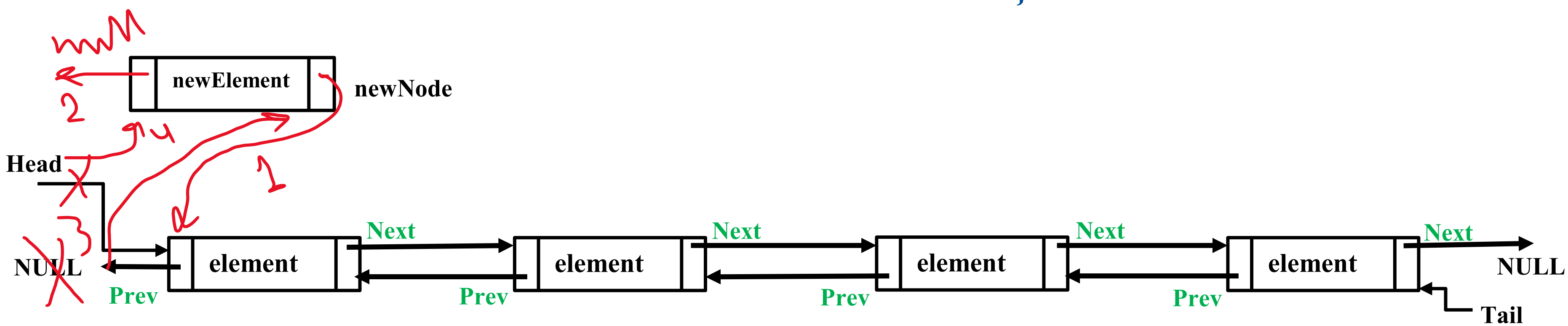
```
// Traversing from tail to the head
public void traverseBackward()
{
  Node current = tail;
  while (current != null) {

  System.out.print(current.data + " ");
  current = current.prev;
  }
}
```



Doubly Linked Lists: Insertion

- At beginning of the list
  - set next of newNode to current head
  - set prev of newNode to null
  - set the prev pointer of the current head to the new node
  - head points to the newNode

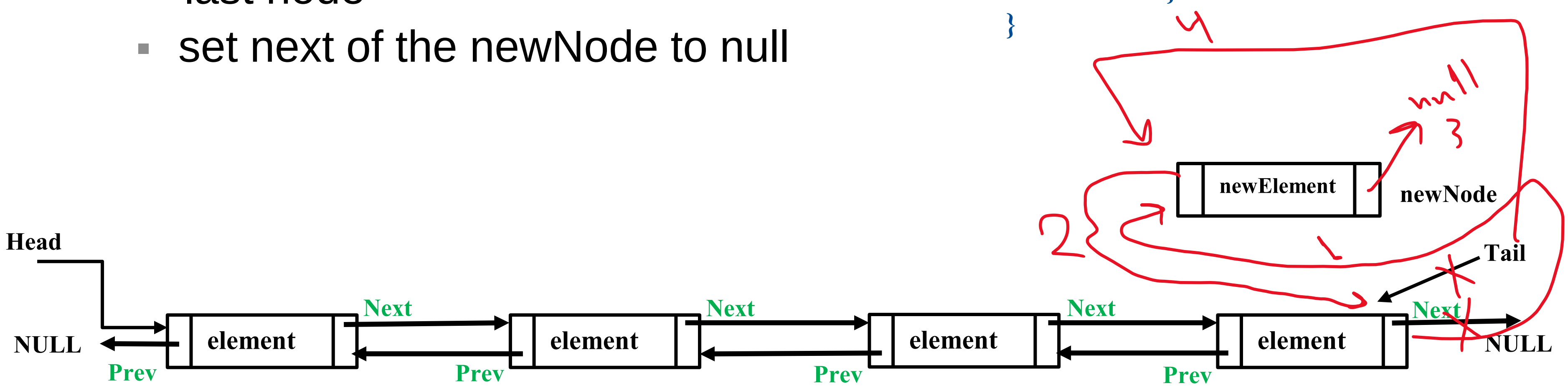
```
public void insertAtBeginning(int data)
{
  Node temp = new Node(data);
  if (head == null) {
  head = temp;
  tail = temp;
   }
  else {
  temp.next = head;
  head.prev = temp;
  head = temp;
 }
}
```



Doubly Linked Lists: Insertion…

- At the end of the list
  - if list is empty, then set head & tail to the newNode
  - if not empty, set the next pointer of current last node to the newNode
  - set Prev of the newNode to the current last node
  - set next of the newNode to null

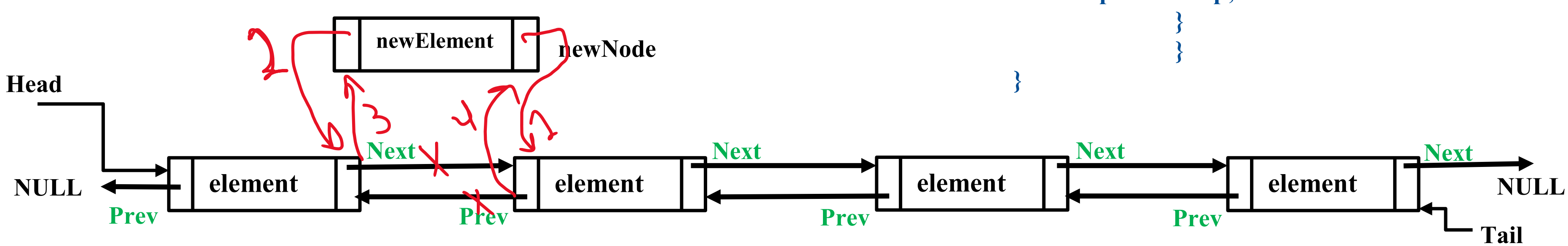
```
public void insertAtEnd(int data)
{
  Node temp = new Node(data);
  if (tail == null) {
  head = temp;
  tail = temp;
  }
  else {
  tail.next = temp;
  temp.prev = tail;
  tail = temp;
          }
}
```



Doubly Linked Lists: Insertion…

- At a specific position in the list
  - set next of newNode to next node of the node at given position
  - set prev of newNode to the node at given position
  - set the next of the node at the given position to the new node
  - if the next node of the newNode not null, set its prev to the newNode

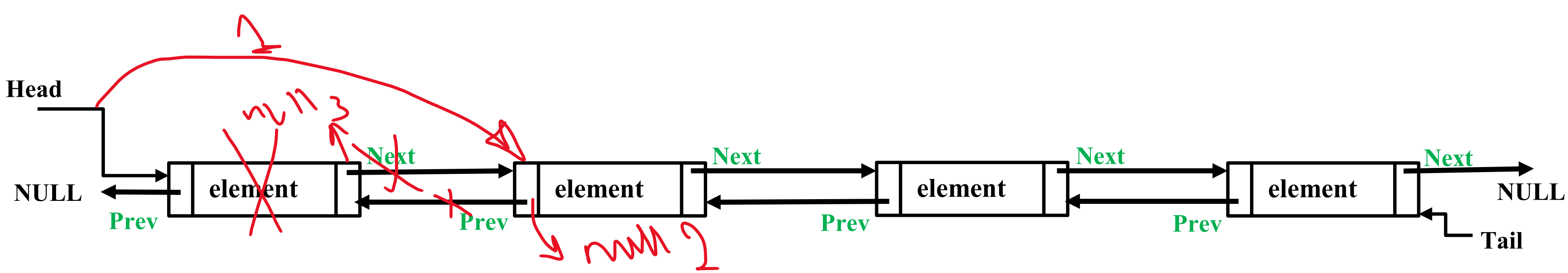
```
public void insertAtPosition(int data, int position)
{
  Node temp = new Node(data);
   if (position == 1) {
   insertAtBeginning(data);
    }
 else {
 Node current = head;
 int currPosition = 1;
while (current != null && currPosition < position) {
current = current.next;
 currPosition++;
 }
if (current == null) {
 insertAtEnd(data);
}else { temp.next = current;
temp.prev = current.prev;
current.prev.next = temp;
current.prev = temp;
                        }
                        }
}
```



Doubly Linked Lists: Deletion

- Delete first node in the list
  - list is empty -> do nothing
  - list has one node -> head & tail = null
  - set head to the next node of the current head
  - set prev of the new head to null
  - set next of the old head to null

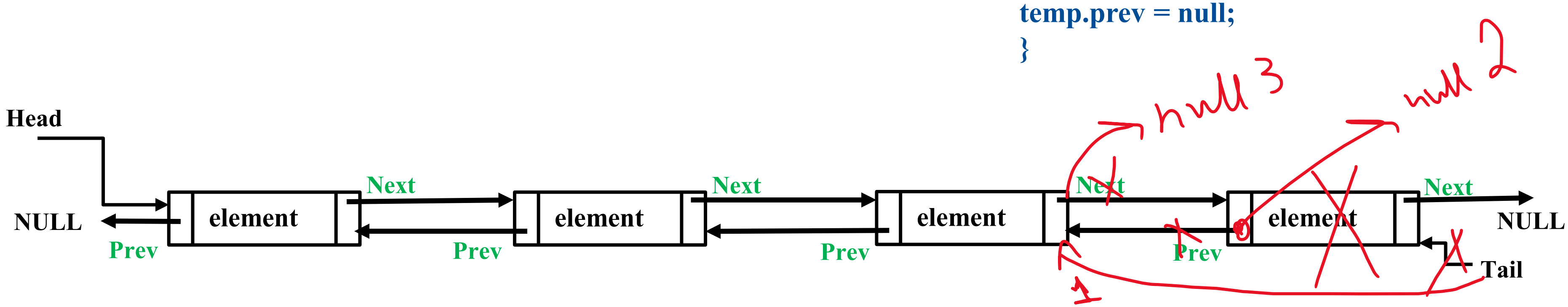
```
public void deleteAtBeginning()
{
  if (head == null) {
  return;
  }

 if (head == tail) {
  head = null;
  tail = null;
  return;
 }

 Node temp = head;
 head = head.next;
 head.prev = null;
 temp.next = null;
}
```



Doubly Linked Lists: Deletion…

- Delete last node in the list
  - list is empty -> do nothing
  - list has one node -> head & tail = null
  - set tail to the previous node of the current tail node in the list
  - set prev of the current tail node to null
  - set the next of the new tail to null

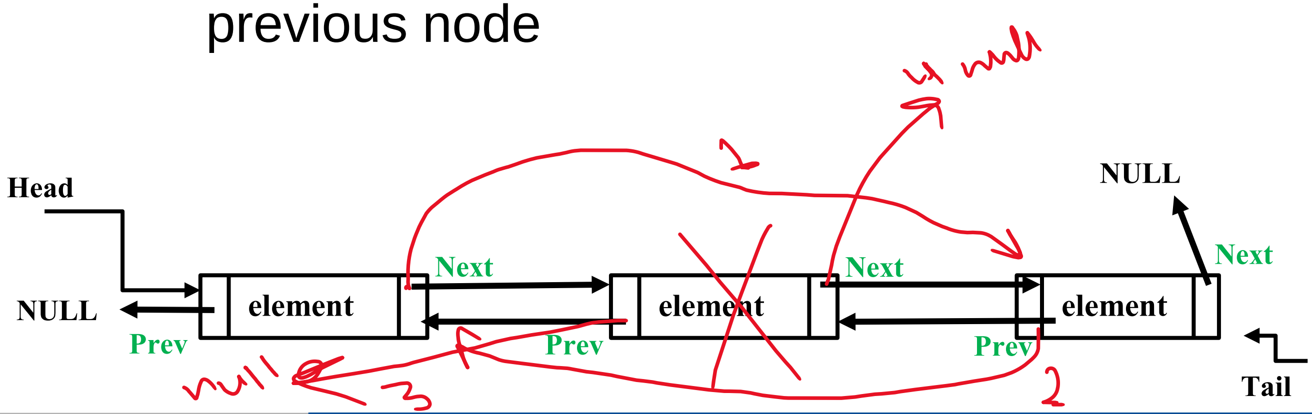
```
public void deleteAtEnd()
{
 if (tail == null) {
 return;
 }

 if (head == tail) {
 head = null;
 tail = null;
 return;
 }

 Node temp = tail;
 tail = tail.prev;
 tail.next = null;
 temp.prev = null;
 }
```



Doubly Linked Lists: Deletion…

- At a specific position in the list
  - traverse to the node at the given position
  - set the next pointer of the previous node to the next node of the node to be deleted
  - if the next node of the node to be deleted is not null, set its previous node to the previous node

```
public void delete(int pos)
{
if (head == null) { return; }
if (pos == 1) {
deleteAtBeginning(); return;
}
Node current = head;
int count = 1;
while (current != null &&
 count != pos) {
current = current.next;
count++;
}
if (current == null) {
System.out.print
("Position wrong");
return;
}
if (current == tail) {
deleteAtEnd();
return;
}
current.prev.next = current.next;
current.next.prev = current.prev;
current.prev = null;
current.next = null;
}
```



Linked Lists: Usage & Applications

- Operating systems
  - scheduling processes and managing system resources
- Database indexing
  - allowing for fast insertion and deletion operations
- Memory management
  - implement memory pools where memory is allocated and deallocated as needed

Linked Lists: Advantages

- Dynamic memory allocation
  - allow for dynamic memory allocation
    - size of the list can change at runtime as elements are added or removed
- Cache friendliness
  - can be cache-friendly as nodes can be stored in separate cache lines, reducing cache misses and improving performance
- Space-efficient
  - are space-efficient, as they only need to store a reference to the next node in each element, rather than a large block of contiguous memory

Linked Lists: Disadvantages

- Poor random-access performance
  - accessing an element in linked lists requires traversing the list from the head/tail to the desired node
  - arrays are better for random access if you know the index
- Increased memory overhead
  - they require additional memory for storing the pointers to the next node in each element
  - extra overhead compared to arrays
- Vulnerability to data loss
  - are vulnerable to data loss if a node’s pointers are lost or corrupted
    - there is no way to traverse the list and access other elements

Assignment: Singly Linked List

- Write two classes to represent the node and list
- In the list class write the following methods
  - a method to traverse the entire list and print all elements
  - three different methods to insert new elements
    - at the head of the list
    - at the tail of the list
    - at a specific position
  - three different methods to delete elements
    - at the head of the list
    - at the tail of the list
    - at a specific position
- Can you write a method to sort the elements in a Singly Linked List?!

Assignment: Doubly Linked List

- Write two classes to represent the node and list
- In the list class, write the following methods
  - two methods to traverse the entire list and print all elements (forward and backward)
  - three different methods to insert new elements
    - at the head of the list
    - at the tail of the list
    - at a specific position
  - three different methods to delete elements
    - At the head of the list
    - At the tail of the list
    - At a specific position
- Can you write a method to sort the elements in a Doubly Linked List?!

Assignment Implemention

- The implementation of the required assignments will be available at the course GitHub repository on Wednesday afternoon
  - Singly Linked Lists

https://github.com/atayeh-israa-university/dataStructures-2023/tree/main/Theory%20-%20EITM2311/Linked%20Lists/Signly%20Linked%20Lists

  - Doubly Linked Lists

https://github.com/atayeh-israa-university/dataStructures-2023/tree/main/Theory%20-%20EITM2311/Linked%20Lists/Doubly%20Linked%20Lists

Thank You!