DATA STRUCTURES (ITPC-203)

Linked Lists

Mrs. Sanga G. Chaki

Department of Information Technology

Dr. B. R. Ambedkar National Institute of Technology, Jalandhar

Contents

- Linked lists
- Doubly linked lists
- Circular linked list
- Representing polynomials using linked lists
- Problems

What is a linked list?

1. Data Structure for storing collection of data
2. Following properties:
   - Successive elements are connected by pointers
   - Last element points to NULL
   - Can grow/shrink in size during execution of a program
   - What is the max length of a linked list? – as long as necessary until memory exhausts
   - It does not waste memory space, but takes extra memory for storing pointers

Linked Lists - ADT

1. Example of primitive datatype?
   - Int, float, char
2. What is an ADT?
   - Abstract data type
   - Mathematical model for a datatype
   - Behavior
     - ➢ Possible values
     - ➢ Possible operations on data of this type
   - Formal definition: a class of objects whose logical behavior is defined by a set of values and a set of operations

Linked Lists - ADT

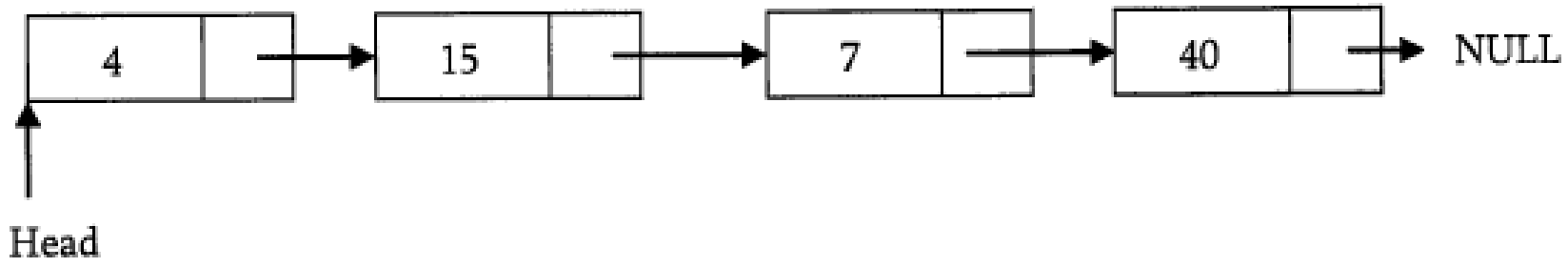
1. Linked list is an ADT
2. Collection of nodes
3. The nodes are themselves data structures with two fields
   - Data
   - Pointer
4. How many data members can be there in a single node?
5. How many max pointer members can be there in a single node?
6. Possible operations:
   - Insertion,
   - Deletion,
   - Traversal
   - Reversal

Linked Lists vs Arrays

1. Array advantages:
   - Simple
   - Faster access to elements – constant access
2. Array disadvantages:
   - Fixed size
   - Difficult to insert elements at a given index – expensive shifting process
3. Linked list advantages:
   - Use of dynamic memory allocation – expansion easy
4. Linked list disadvantages:
   - Access time to individual elements
   - Storage of pointers take space

Singly Linked Lists

1. Collection of nodes
2. Each node has a next pointer to the following element
3. Last node points to NULL

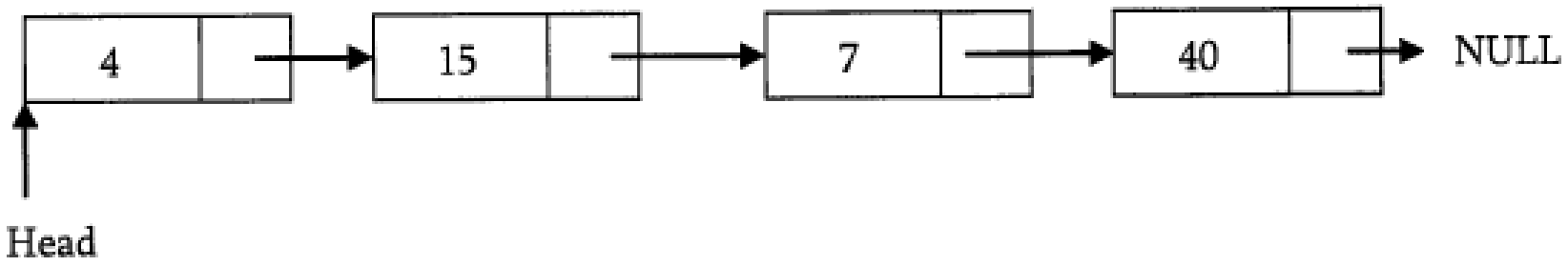


4. Node declaration for a linked list of integers:

```
struct ListNode {
        int data;
        struct ListNode *next;
};
```

Singly Linked Lists: Basic Operations : Traversal

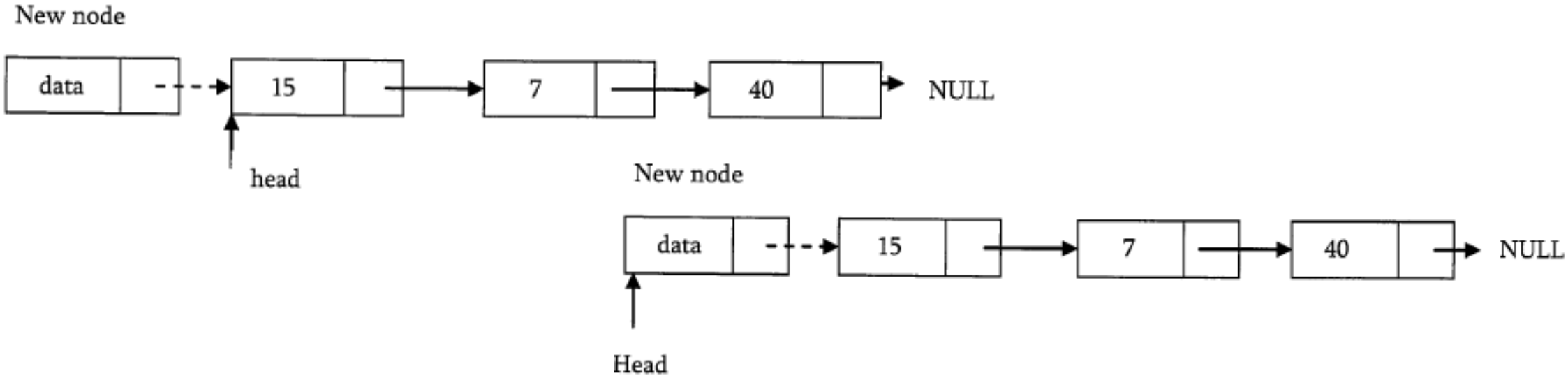
1. Assume head points to the first node
2. Follow the next pointers
3. Display the contents of the nodes as they are traversed/Count nodes
4. Stop when the next pointer points to NULL



5. What is the time complexity?
   - O(n) for scanning complete list of size n

Singly Linked Lists: Basic Operations : Insertion

1. Three cases:
   - Insertion at the beginning
   - Insertion at the end
   - Insertion at a random location
2. Insertion at the beginning:
   - Modify only 1 next pointer
   - Update the next pointer of the new node to point to the current head
   - Update head pointer to point to the new node



Singly Linked Lists: Basic Operations : Insertion

1. Insertion at the end:
   - Modify 2 next pointers – last node and new node
   - New node's next pointer points to NULL
   - Last node's next pointer points to new node
2. Insertion at a random location
   - Modify 2 next pointers
   - Let us have to insert at n^{th} location.
   - Traverse up to the $(n-1)^{th}$ location.
   - New node's next pointer points to the location where $(n-1)^{th}$ node's next pointer was pointing.
   - $(n-1)^{th}$ node's next pointer points to new node

Singly Linked Lists: Deletion - Recap

1. Deletion at the beginning
2. Deletion at the end:
3. Deletion at a random location

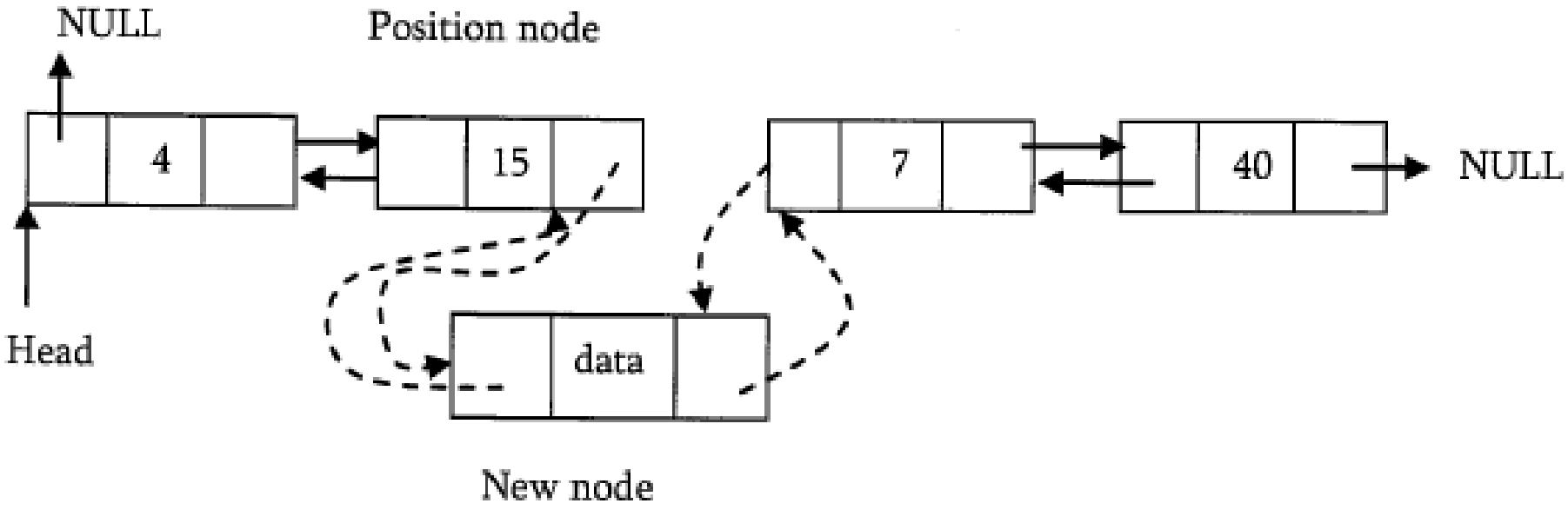
Doubly Linked Lists

1. Given a node we can navigate in both directions
2. We can delete a node even if do not have previous node's address
3. Each node has two pointers – occupy more space
4. Insertion/deletion of nodes take more time – more pointer operations

```
struct DLLNode {
        int data;
        struct DLLNode *next;
        struct DLLNode *prev;
};
```

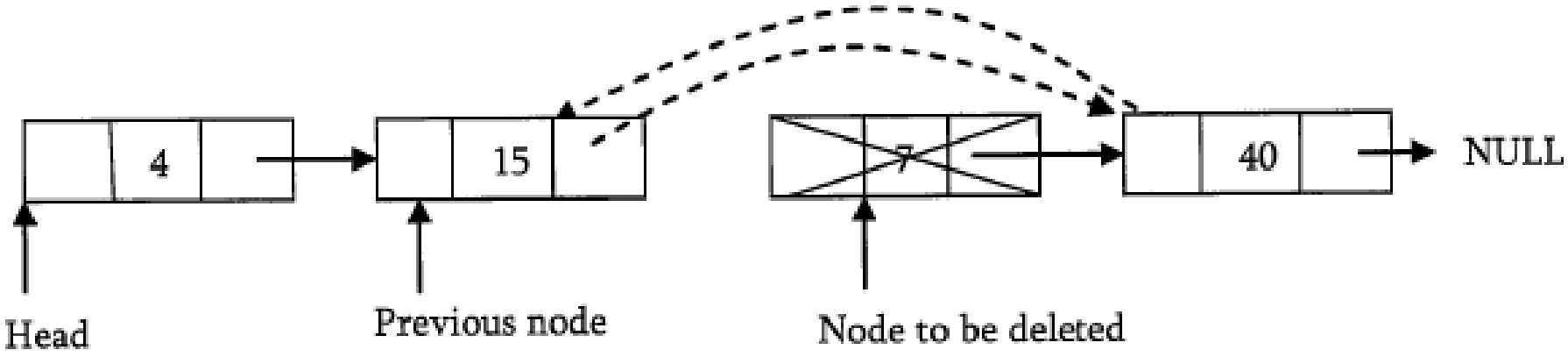
Doubly Linked Lists - Insertion

1. Insertion at
   - The beginning – no of pointer operations = 3
   - The end – no of pointer operations = 3
   - In the middle – no of pointer operations = 4



Doubly Linked Lists - Deletion

1. Deletion at
   - The beginning
   - The end
   - In the middle – How many pointer operations = 2



Polynomials and Linked Lists

1. Representing Polynomials As Singly Linked Lists
2. Manipulation of symbolic polynomials
3. Able to represent any number of different polynomials as long as memory is available
4. We draw poly-nodes as:

5. Lets take an example:

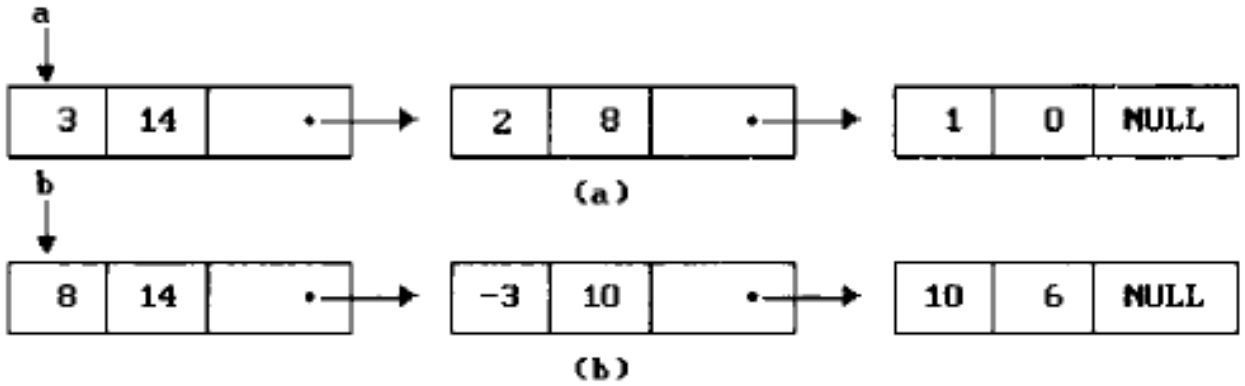
$$a = 3x^{14} + 2x^{8} + 1$$

and

$$b = 8x^{14} - 3x^{10} + 10x^{6}$$

Polynomials and Linked Lists

1. In linked list format



2. Now that we know the representation, we can manipulate these polynomials

Polynomials Addition

1. We examine the terms of the polynomials starting at the first nodes of the two linked lists a and b.
2. If the exponents of the two terms are equal, we add the two coefficients and create a new term for the result linked list.
3. We also move the pointers to the next nodes in a and b.
4. If the exponent of the current term in a is less than the exponent of the current term in b, then we create a duplicate term of b, attach this term to the result.
5. We advance the pointer to the next term in b.
6. We take a similar action on a if a->expon > b—>expon.

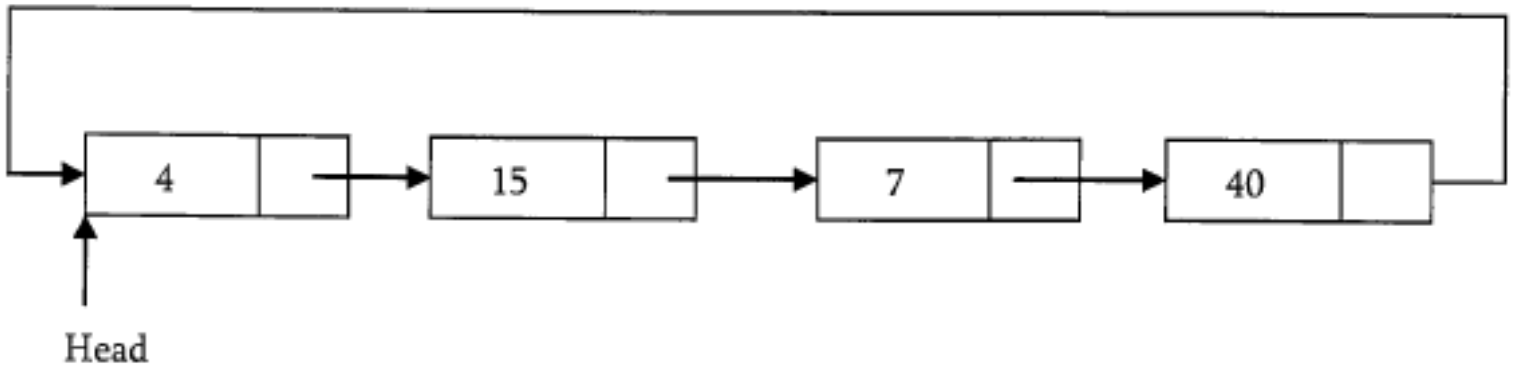
Polynomials Addition

1. What are the three cost-incurring operations for this algorithm?
   - coefficient additions
   - exponent comparisons
   - creation of new nodes for d

Circular Linked List

Circular Linked Lists

1. The circular linked list is a linked list where all nodes are connected to form a circle.
2. No end – no node points to NULL
3. The next pointer of the last node points to the first node



4. Traversal, insertion, deletion

Circular Linked Lists

1. Circular singly linked list: In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list.
2. We traverse the circular singly linked list until we reach the same node where we started.
3. Circular Doubly linked list: Circular Doubly Linked List has properties of both doubly linked list and circular linked list
4. two consecutive elements are linked or connected by the previous and next pointer and
5. the last node points to the first node by the next pointer and
6. also the first node points to the last node by the previous pointer.

Operations on Circular Singly Linked Lists

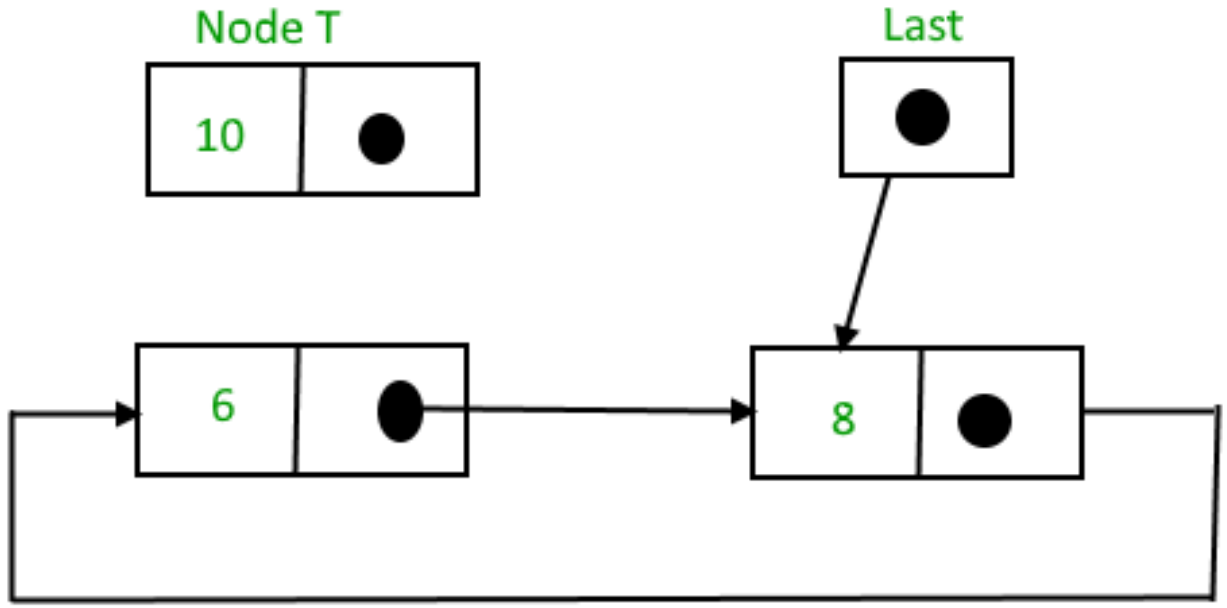
```
struct Node {
        int data;
        struct Node *next;
};
```

```
//Pseudocode for creating a circular LL
Node* one = createNode(3);
Node* two = createNode(5);
Node* three = createNode(9);

// Connect nodes
one->next = two;
two->next = three;
three->next = one;
```

Circular Singly Linked Lists - Insertion

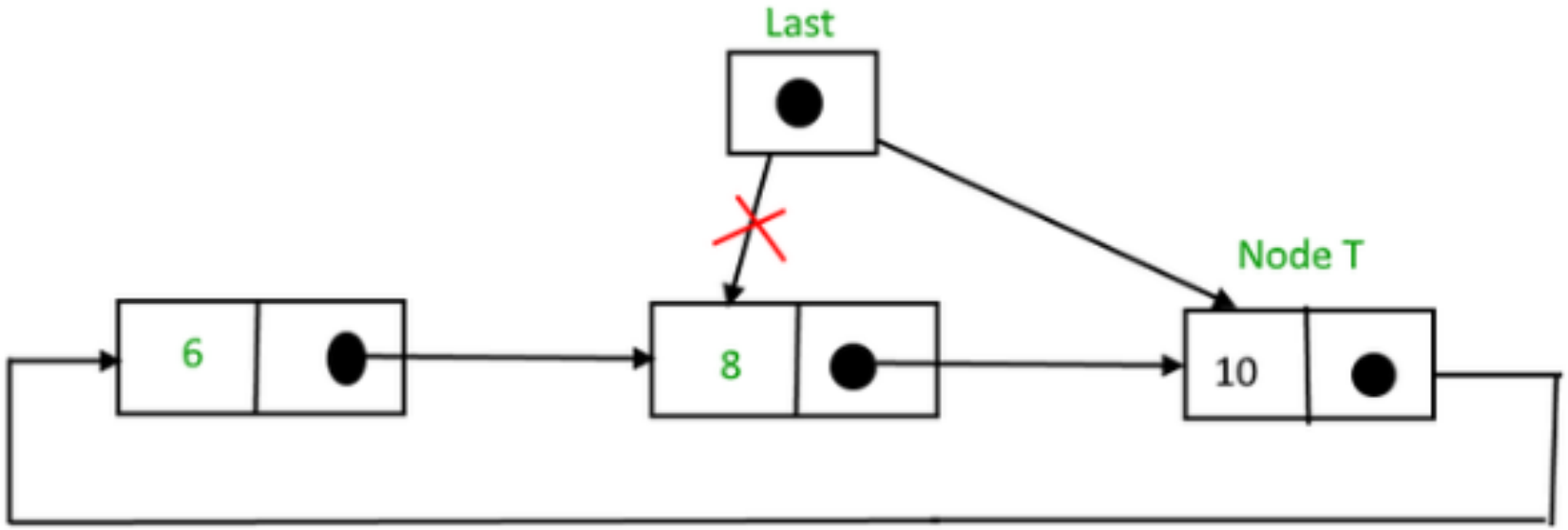
1. Insertion at the beginning of the list:
2. To insert a node at the beginning of the list, follow these steps:
   - Create a node, say T.
   - Make T -> next = last -> next.
   - last -> next = T.



Circular linked list before insertion

Circular Singly Linked Lists - Insertion

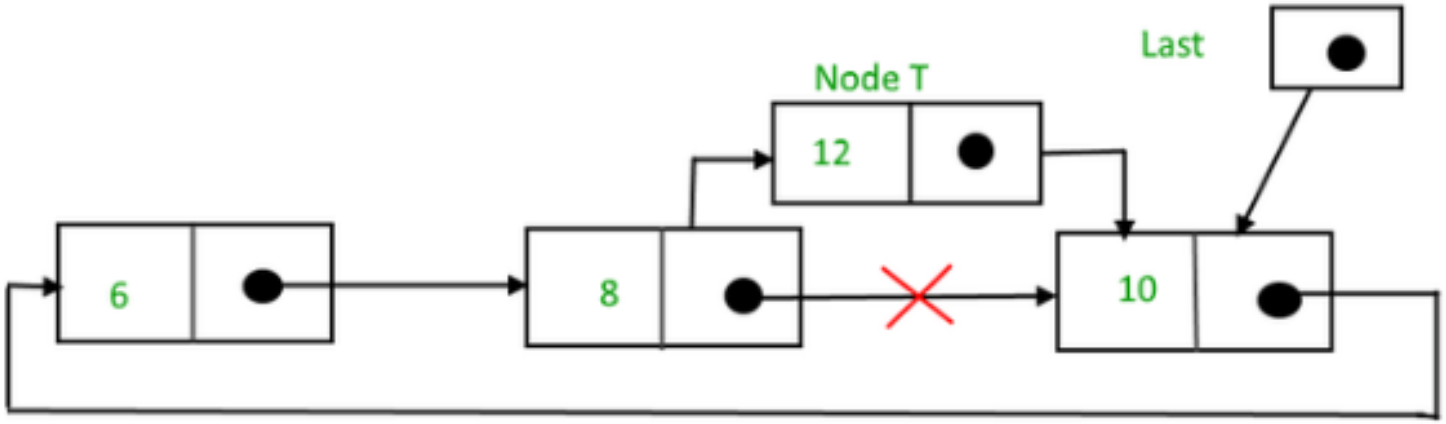
1. Insertion at the end of the list: To insert a node at the end of the list, follow these steps:
   - Create a node, say T.
   - Make T -> next = last -> next;
   - last -> next = T
   - last = T



Circular linked list after insertion of node at the end

Circular Singly Linked Lists - Insertion

1. Insertion in between the nodes: To insert a node in between the two nodes, follow these steps:
   - Create a node, say T.
   - Search for the node after which T needs to be inserted, say that node is P.
   - Make T -> next = P -> next
   - P -> next = T.



Circular linked list after insertion

Operations on Circular Singly Linked Lists

1. Similarly, deletion is also possible.
2. Can there be a circular linked list with one node = yes, points to itself.
3. Applications/Advantages:
   - Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
   - Circular lists are useful in applications to repeatedly go around the list.
4. Disadvantages:
   - Reversing will be difficult
   - It is possible for the code to go into an infinite loop if it is not handled carefully.

Problems

1. Given a linked list of type integer, size m, find the n^{th} node from the end of a linked list. What is the time complexity?
2. Can there be a better approach to the previous problem, with time complexity O(n)?
3. Insert a node in a sorted linked list
4. Reverse a singly linked list
5. Find the middle of the linked list. How many scans do you need? Can it be done in one scan?
6. Write a recursive function to count the number of nodes in a linked list