# 19CSE 111: Foundations of Data Structures

## Lecture 8:Linked Lists

Dr. Vidhya Balasubramanian



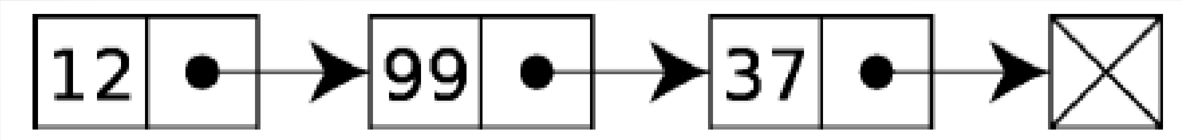
Amrita School of Engineering
Amrita Vishwa Vidyapeetham

# The concept

- Currently we have seen array based implementations
- Limitations of Arrays
  - The size is bounded
  - Results in too many resize operations or wastage of memory
- Solution
  - Dynamically allocate and deallocate memory as and when data is added and removed

# Dynamically Allocating Elements

- Allocate elements one at a time
  - Each element keeps track of next element
- Results in a linked list of elements
  - Elements track next element with a pointer
  - elements can easily be inserted or removed without reallocation or reorganization of the entire structure

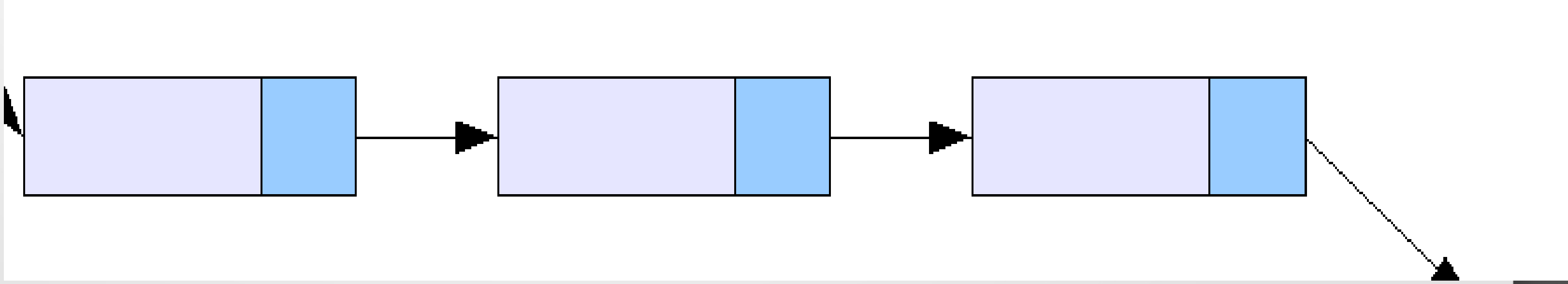


# Linked Lists

- Developed in 1955-56 by Allen Newell, Cliff Shaw and Herbert A. Simon at RAND Corporation as the primary data structure for their Information Processing Language (IPL)
- Must have the following
  - Way to indicate end of list
    - NULL pointer
  - Indication for the front of the list
    - Head Node
  - Pointer to next element

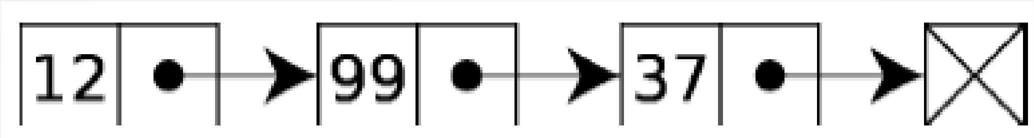
# Linked Lists: Basic Concepts

- Each record of linked list is an element or a node
- Each node contains
  - Data member which holds the value
  - Pointer "next" to the next node in the list
  - Head of a list is the first node
  - Tail is the last node
- Allows for insertion and deletion at any point in the list without having to change the structure
- Does not allow for easy access of elements (must traverse to find an elt)

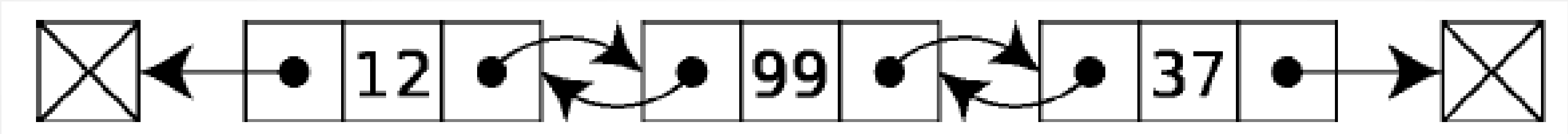


# Linked Lists: Types

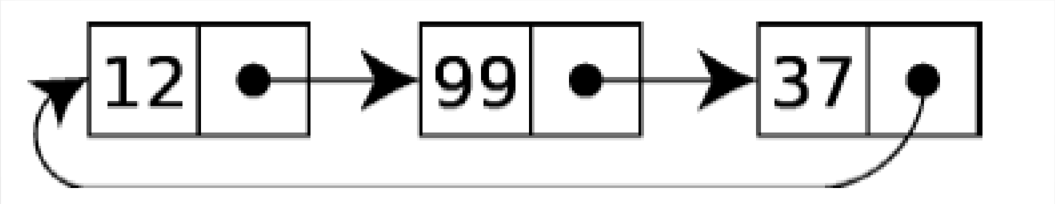
- Singly linked list




- Doubly linked list

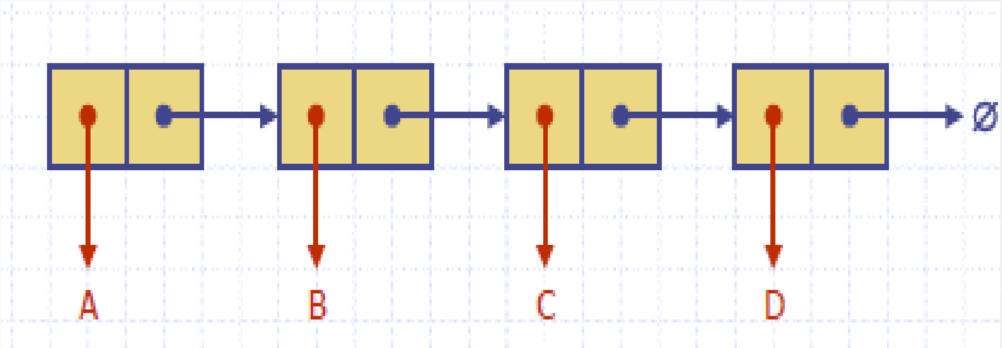



- Circular linked list



# Singly Linked Lists

- Keeps elements in order
  - Uses a chain of next pointers
  - Does not have fixed size, proportional to number of elements
- Node
  - Element value
  - Pointer to next node
- Head Pointer
  - A pointer to the header is maintained by the class



# Basic Linked List Definition

- class Node():

  element // The data being stored in the node

  next // A reference to the next node, null for last node, of the type Node

- class List():

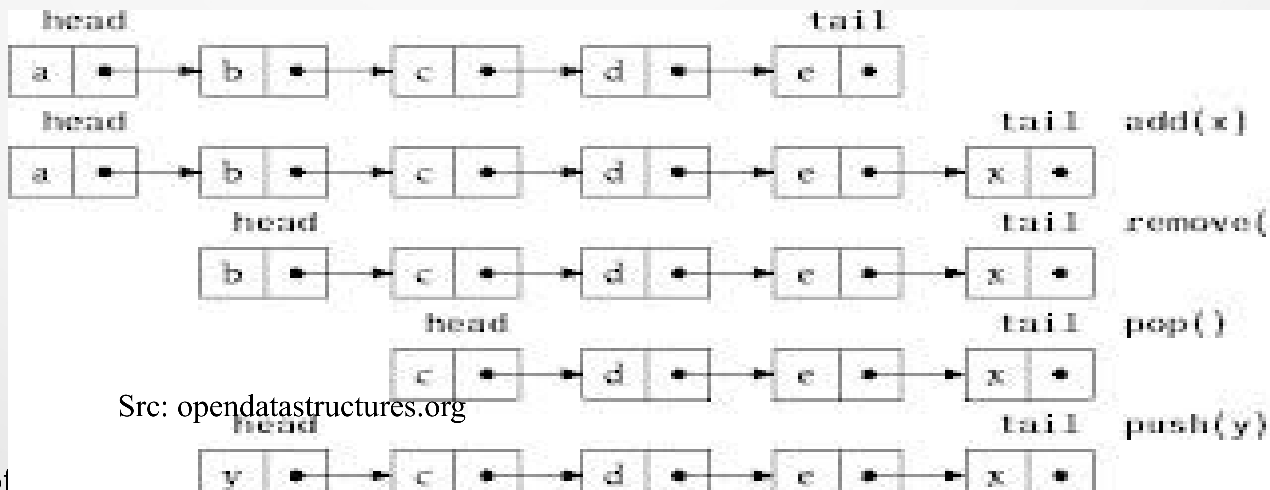
  Node firstNode

  // points to first node of list; null for empty list

  // this is also known as the head

# Insertion and Deletion

- Insertion can be at head or tail
  - Create new node, and make new node point to head, and make it the new head
  - If using tail pointer, point next of tail to new node, and next of new node to null
- Deletion
  - Requires the reorganization of next pointers



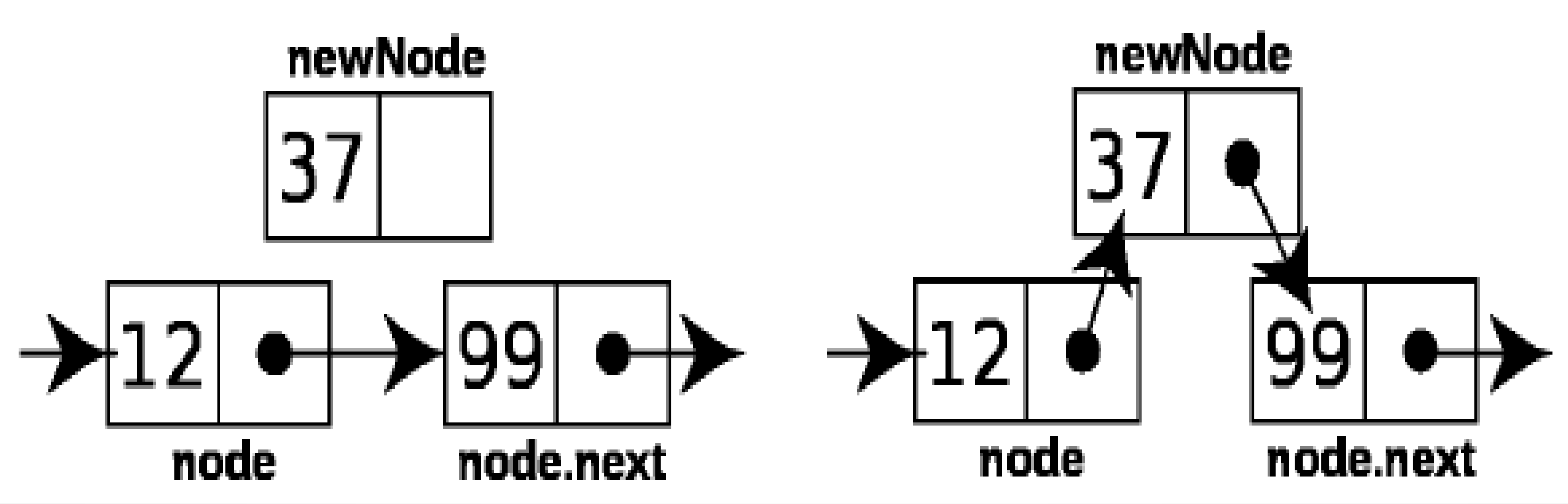

Src: opendatastructures.org

# List ADT: Functions

- **Algorithm** insertAfter(Node node, Node newNode) // insert newNode after node

    newNode.next ← node.next

    node.next ← newNode



# List ADT Functions:

- **Algorithm** insertFirst(List list, Node newNode)

  // insert node before current first node

      newNode.next := list.Head

      list.Head :=  newNode

- **Algorithm** insertLast(List list, Node newNode)

  // insert node after the current tail node

      tail.next ← newNode

      newNode.next ← NULL

# List ADT: Delete Functions

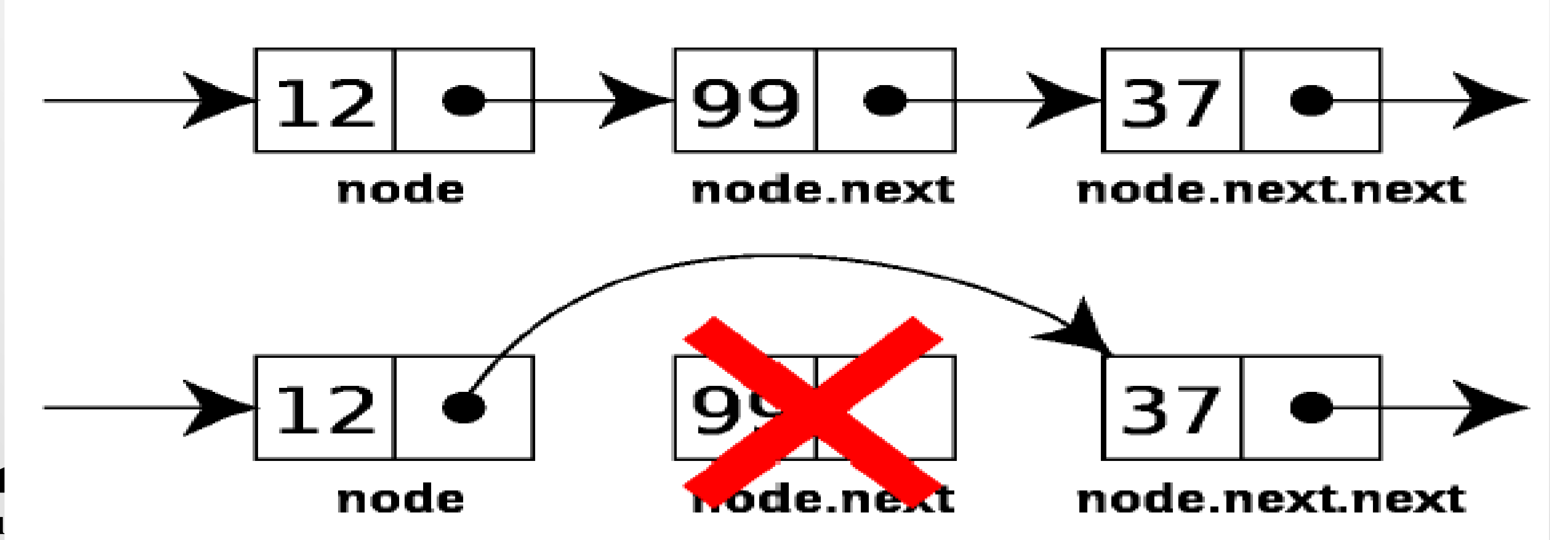
- **Algorithm** removeAfter(Node node) // remove node past this one

  obsoleteNode ← node.next

  node.next ← node.next.next

  ObsoleteNode.next ← null

  destroy obsoleteNode



# Traversing the list

- Algorithm Traverse()
  - Node ← list.firstNode
  - while node not null

    do something with node.element

    node ← node.next

# Other possible list functions

- first() : return the first node of the list, error if S is empty
- last(): return last node of the list, error if S is empty
- isFirst(*p*): returns true if *p* is the first or head node
- isLast(*p*): returns true if *p* is the last node or tail
- before(*p*): returns the node preceding the node at position *p*
- *getNode(i): return the node at position i*
- after(*p*): returns the node following the node at position *p*
- size() and isEmpty() are the usual functions

# List: Update Functions

- replaceElement(*p*,*e*): Replace element at node at *p* with element *e*
- swapElements(*p*,*q*): Swap the elements stored at nodes in positions *p* and *q*
- insertBefore(*p*,*e*) Insert a new element *e* into the list S before node at *p*

# Complexity Analysis

- Time Complexity
  - size – $O$(n)
  - isEmpty – $O$(1)
  - first(), isFirst(), isLast()– $O$(1)
  - insertAfter(*p,e*), after(*p*) – O(1) (if pointer to p given)
- Space Complexity
  - $O$(n)

# Exercises

- Give an algorithm for finding the penultimate node in a singly linked list where the last element is indicated by a null next pointer
- Give an algorithm for concatenating two singly linked lists L and M, with header nodes, into a single list L' where
  - L' contains all nodes of L in their original order followed by all nodes of M (in original order)
  - What is the running time of your algorithm if n is the number of nodes in L, and m is the number of nodes in M?

# Stack: Linked List Based Implementation

- Top element is stored as the head (first node) of the linked list
- Insertion and deletion always at the front
- The stack class has the following variables
  - Node topnode //top is the head node
    - Initialized to NULL
  - sz //variable to keep track of the size of the list
    - initialized to 0

# Stack ADT Functions

- **Algorithm** size()

  **return sz**

- **Algorithm** isEmpty()

  **return** (*sz* == 0)

- **Algorithm** top()

  if isEmpty() **then**

      **throw** a StackEmptyException

  **return** *topnode.element*

# Stack ADT Functions

- **Algorithm** push(*o*)

  **if** size() = *N* **then**

  **throw** a StackFullException

  *newNode* ← *new Node(o, topnode)*

  *topnode* ← *newNode*

  *sz++*

# Stack ADT Functions

- **Algorithm pop**()

  if isEmpty() **then**

      **throw** a StackEmptyException

  *Node oldNode* ← *topnode*

  *topnode* ← *topnode.next*

  *sz--*

  *o* ← *oldNode.element*

  *delete oldNode*

  **return** *o*

# Queue: Linked List Based Implementation

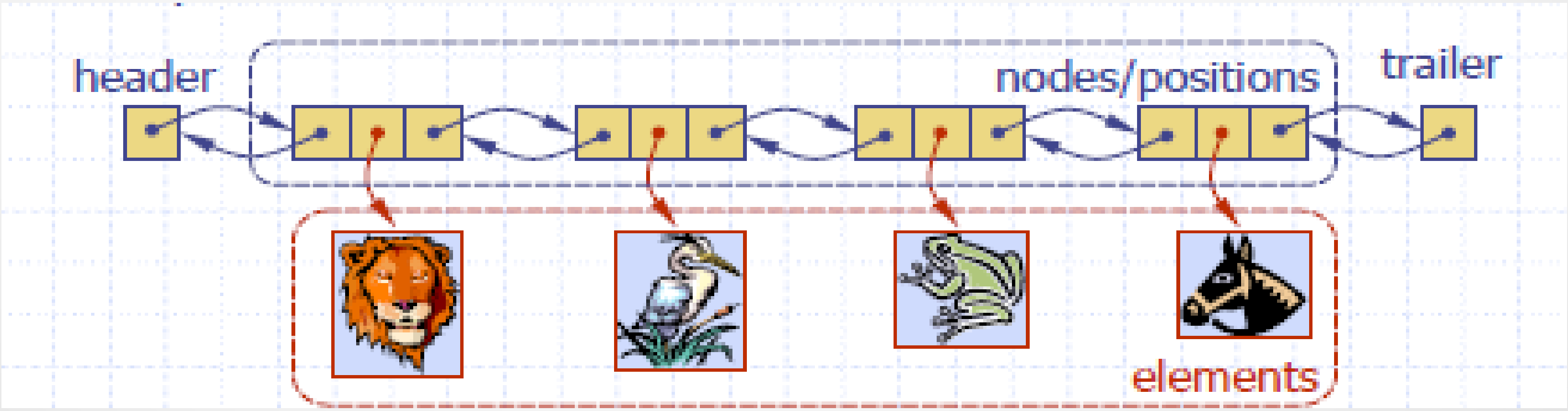
- Can be done similarly
- Here insertion is done at the tail
- Deletion is at the head

# Exercises

- Design and implement an SLList method, secondLast(), that returns the second-last element of an SLList. Do this without using the member variable, n, that keeps track of the size of the list.
- Describe and implement the following List operations on an SLList
  - get(i) // get the node at position i
  - set(i,x) // set the value of node at ith position to x
  - add(i,x) // add a node with value x with position i
  - remove(i). //remove node at position i
  - Each of these operations should run in O(1 + i) time.

# Doubly Linked List

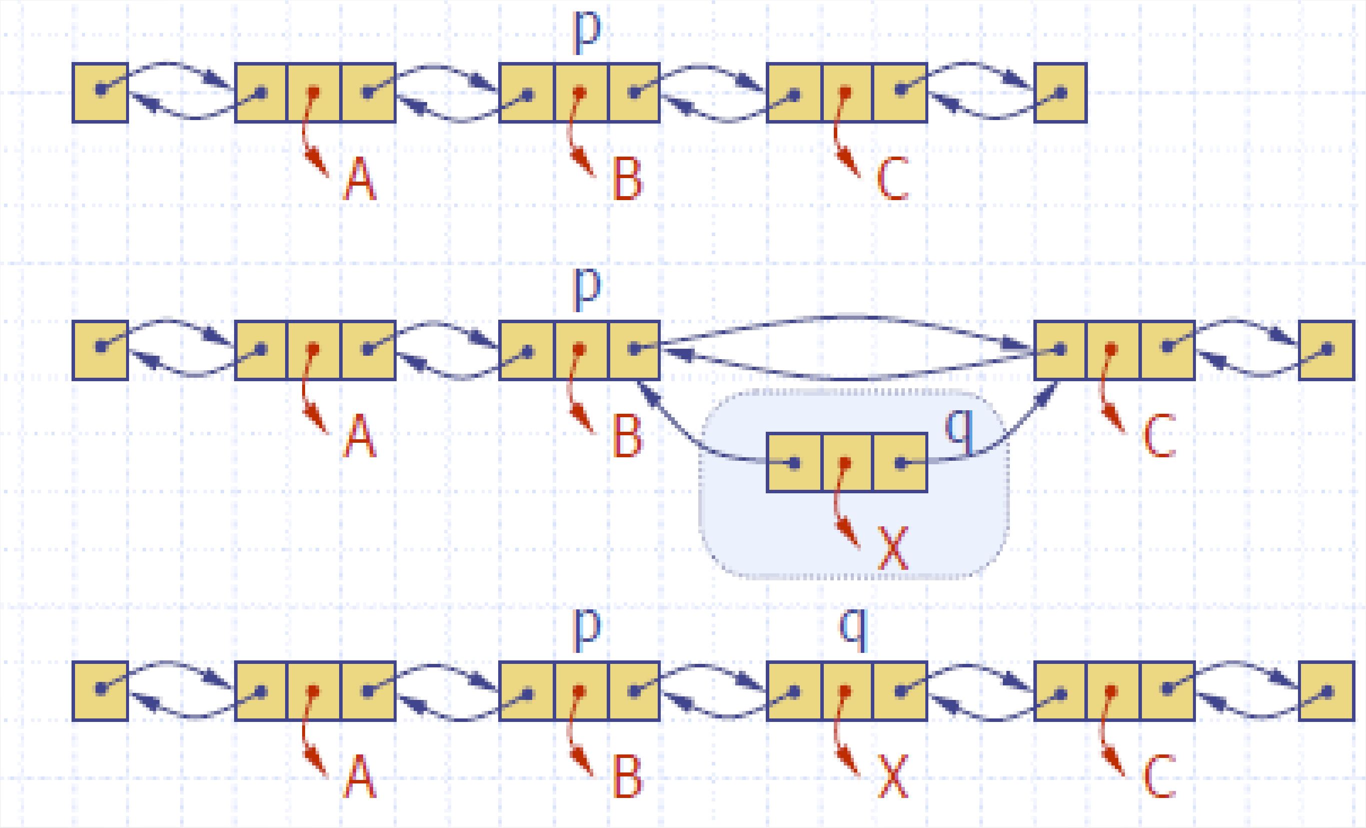
- Nodes implement the position and store the following
  - Element
  - Link to previous node
  - Link to next node
- Trailer and Header nodes




Src: Goodrich notes

# Insertion: Doubly Linked List

- insertAfter(p,X)



# InsertAfter

- **Algorithm** insertAfter (Node p, Value x) // insert newNode before node

  q ← new Node()

  q.value ← x

  q.next ← p.next

  q.prev ← p

  q.next.prev ← q

  q.prev.next ← q

  n++; //increment size

# InsertBefore

- **Algorithm** insertBefore (Node r, Value x) // insert newNode before node

  q ← new Node()

  q.value ← x

  q.prev ← r.prev

  q.next ← r

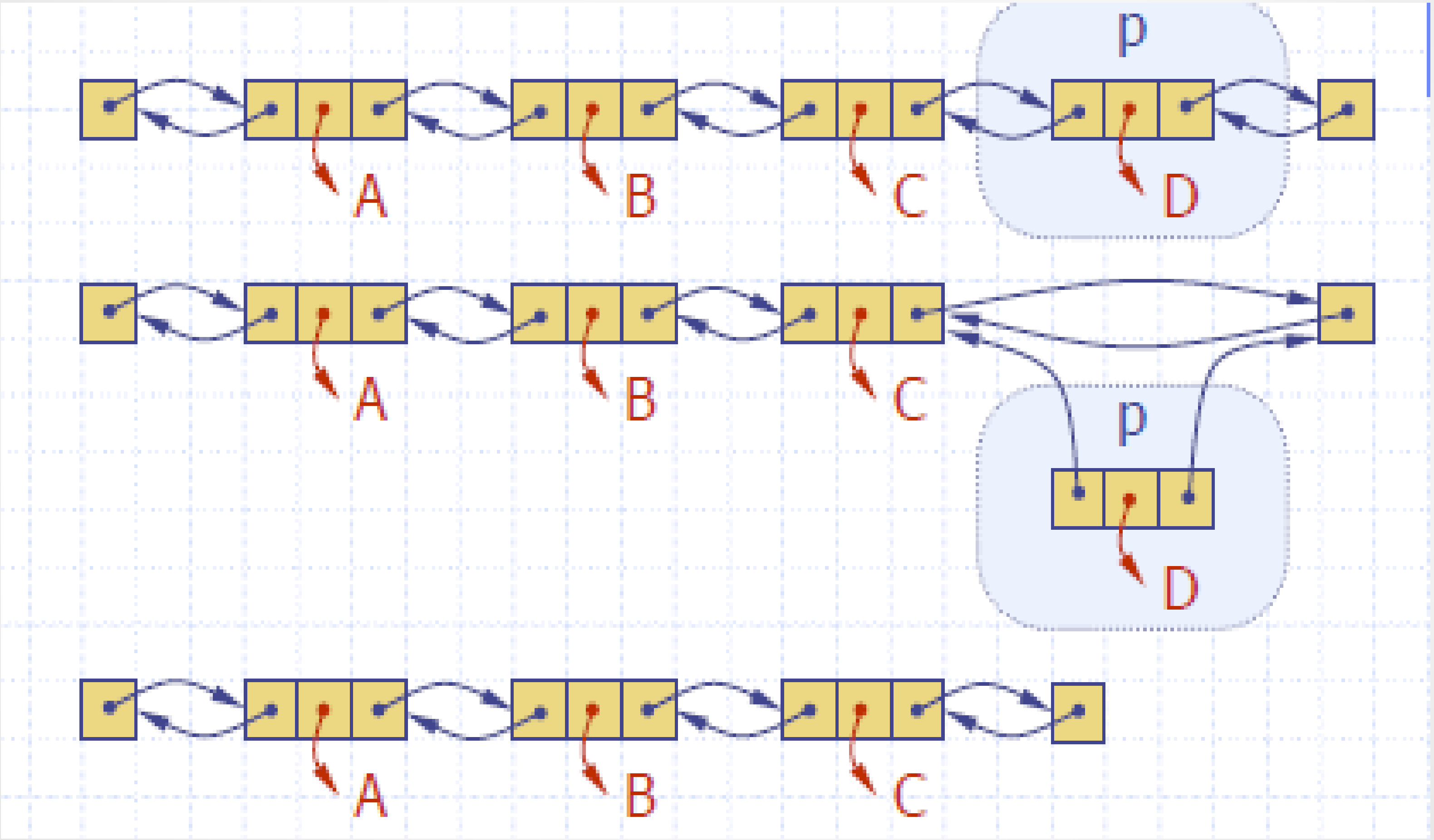
  q.next.prev ← q

  q.prev.next ← q;

  n++; //increment size

# Deletion: Doubly Linked List

- remove(p)



# Remove Operation

- **Algorithm** remove(Node *p*)

  $p$.prev.next $\leftarrow$ $p$.next

  $p$.next.prev $\leftarrow$ $p$.prev

  delete $p$

# Getting a Node given a location

- **Algorithm** getNode(*p*)

```
Node tnode
if (p < n / 2)
    tnode ← head.next;
    for (i = 0; i < p; i++)
        tnode ← tnode.next
    else
        tnode ← tailnode
        for (i = n; i > p; i--)
        tnode ← tnode.prev
return (tnode)
```

# Doubly Linked List vs Singly Linked List

- Doubly linked list requires more space per node
  - Elementary operations more expensive
- Allow sequent access in both directions
  - one can insert or delete a node in a constant number of operations given only that node's address
  - in a singly linked list, one must have the address of the pointer to that node or the link field in the previous node

# Circular Linked List

- If in a linked list, the tail points to the head, it is a circular linked list
  - Can be for both singly or doubly linked list
- Works for arrays that are naturally circular
  - Representing points in a polygon
  - Processes to be scheduled in round robin order
- Supports access to both ends of the list without using extra pointers
- Can traverse the full list from any node

# Exercises

- Give an algorithm to merge two doubly linked lists L and M into one list. What is the running time of your algorithm.
- Give a pseudocode of an algorithm to swap two nodes x, and y in a singly linked list given pointers only to x and y.
  - Do the same for the case of a doubly linked list
- Describe in pseudocode a linear-time algorithm for reversing a singly linked list L, so that the ordering of the nodes becomes exactly opposite of what it was before.