Data Structures and Analysis of Algorithms
CST 225-3

Linked List

Limitations of an Array

- Arrays are of fixed length.
- After creating an array to change the size, it requires creating a new array and copy all data from old sized array to new sized array.
- Data in an array are arranged in a sequential manner.
- To insert an element in between two consecutive elements of an array, some elements have to be shifted to a side.
- As a solution **Linked Structures** can be used.

Linked Structures

- A linked list is a linear data structure.
- The elements in a linked list are linked using references.
- Consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

More Terminology

- A node's **successor** is the next node in the sequence.
  The last node has no successor.

- A node's **predecessor** is the previous node in the sequence.
  The first node has no predecessor.

- A list's **length** is the number of elements in it.
  A list may be empty (contain no elements).

Linked Structures

- There can be different types of Linked structures like;
  - ✓Singly Linked List
  - ✓Doubly Linked List
  - ✓Circular Linked List

Linked List

- A Linked List is called ”linked” because each node in the series has a pointer(reference) that points to the next node in the list.
- Head: Pointer to the first node.
- The last node points to null.

Advantages of Linked Lists

- Dynamic in nature where memory allocation is done during execution time.
- Linked lists provide flexibility in allowing the items to be rearranged efficiently.
  - Insert an element.
  - Delete an element.

Disadvantages of Linked Lists

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.

Applications of Linked Lists

The following are some of the most common applications of using Linked Lists.

- Implementing stacks, queues and graphs
- Implementing the undo functionality of software

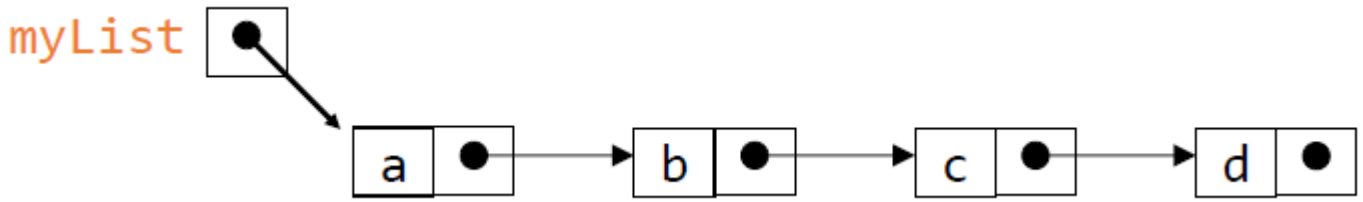
Arrays vs Linked Structures

Singly Linked Lists

- A linked list that navigates only to the forward direction through the data elements.
- Consists of nodes where each node contains a data field and a reference(link) to its the next node in the list.
- The operations that can be performed on singly lists are insertion, deletion and traversal.

Singly Linked Lists

- Each node contains a value and a link to its successor (the last node has no successor)
- The header points to the first node in the list (or contains the null link if the list is empty).



- We can traverse from one node to other node only in one direction and cannot traverse back.

Singly Linked Lists-Drawbacks

- Can traverse from one node to other node only in one direction and cannot traverse back.

ListNode Declaration

```
Class ListNode
{
    Int data;
    ListNode next;
    ListNode() {
        this.data = 0;
        next = NULL;}
    ListNode(int givenData) {
        this.data = givenData;
        next = NULL;}
}
```

Inserting a new node

Possible cases

1. Insert into an empty list
2. Insert in front of the list
3. Insert at back of the list
4. Insert in middle of the list

Insert node at the start of list

- **Pseudo code:**

new_node → next = head;

head = new_node;

Insert node in the middle of the list

- **Pseudo code:**

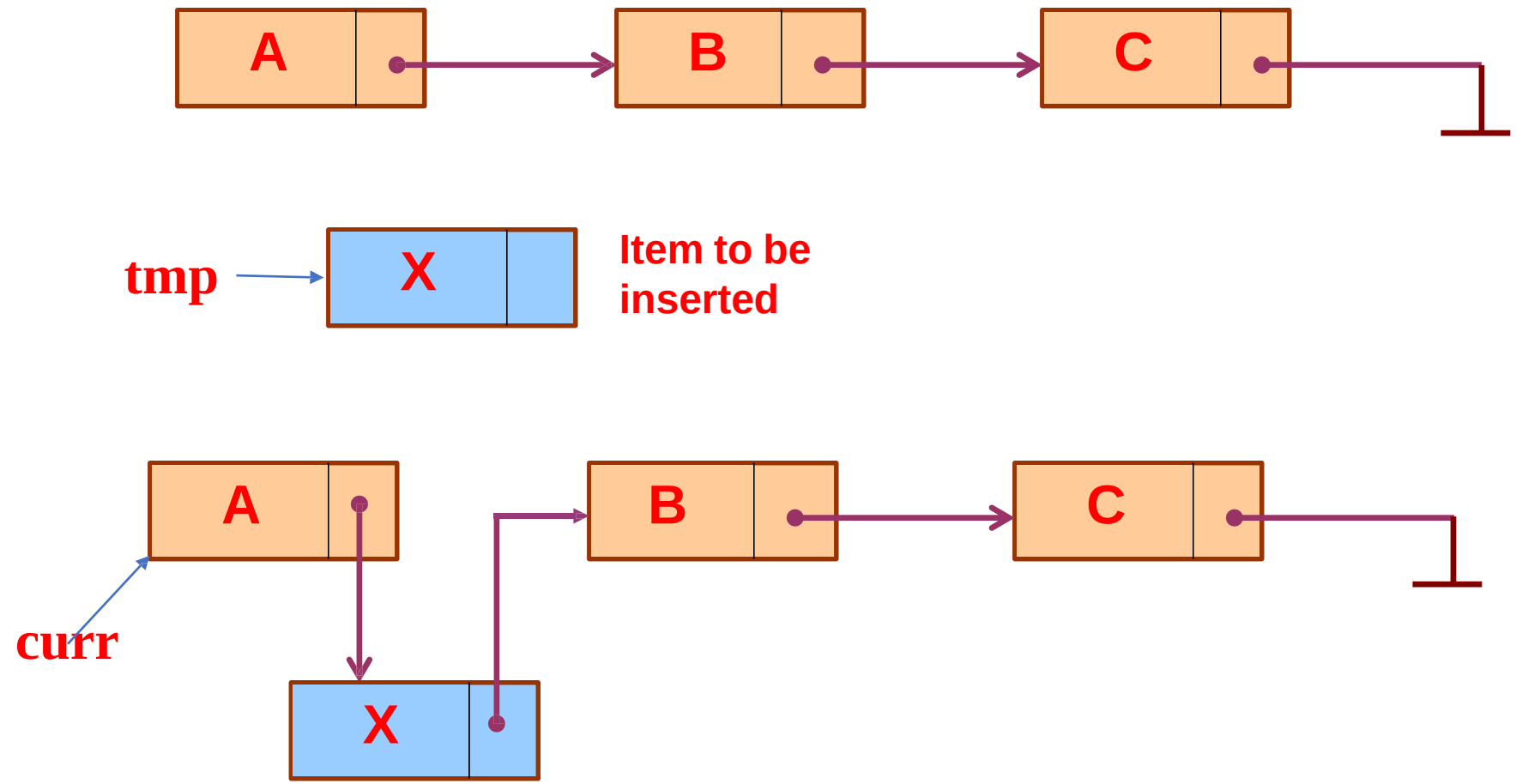
```
While (current! = insert.position)
{
current = current → next;
}
store_next = current → next;
current → next = new_node;
new_node → next = store_next ;
```

Insert a node at the end of list

- **Pseudo code:**

```
While (current → next != NULL)
{
current = current → next;
}
current → next = new_node;
New_node → next = NULL;
```

Node Insertion - General Scenario



Insertion General Case

- For insertion: (pseudocode)

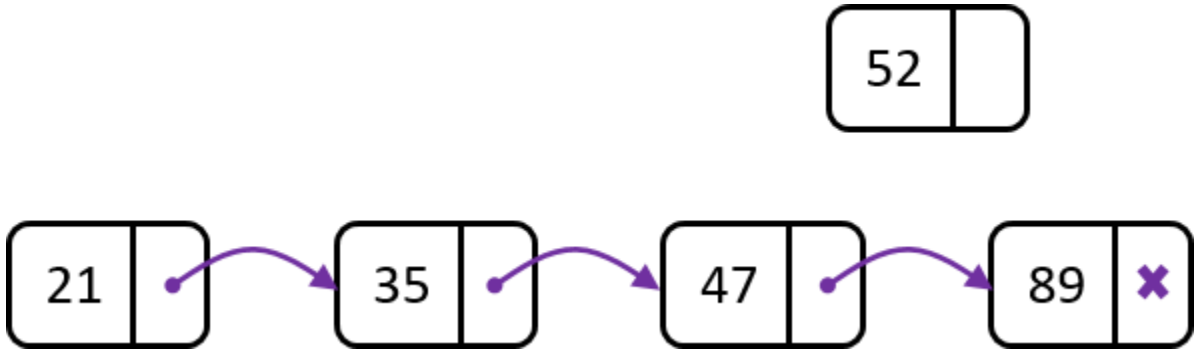
  - *Create a new node.*
    Store data in the new node.
    If there are no nodes in the list
    Make the new node the first node.
    Else
    Traverse the List to Find the place for the insertion.
    Add the new node to the list.
    End If.

Operations of Linked Lists

- The following basic operations are supported by Linked Structures.
  - Insertion() – adding an element at the beginning of the list
  - Deletion() – deletes an element from the beginning of the list
  - Display() – displays the complete set of elements
  - Search() – searches an element using a given key
  - Delete() – deletes an element using a given key

Inserting a value to a Singly Linked List

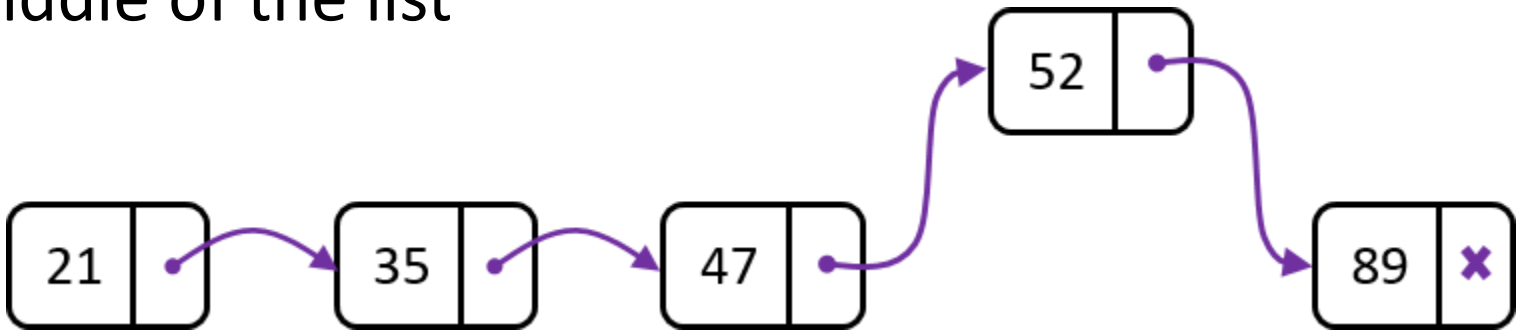
Insert in middle of the list



Memory Address	Data Element	Pointer
0	21	1
1	35	2
2	47	3
3	89	null

Inserting a value to a Singly Linked List

Insert in middle of the list



Memory Addressc	Data Element	Pointer
0	21	1
1	35	2
2	47	4
3	89	null
4	52	3

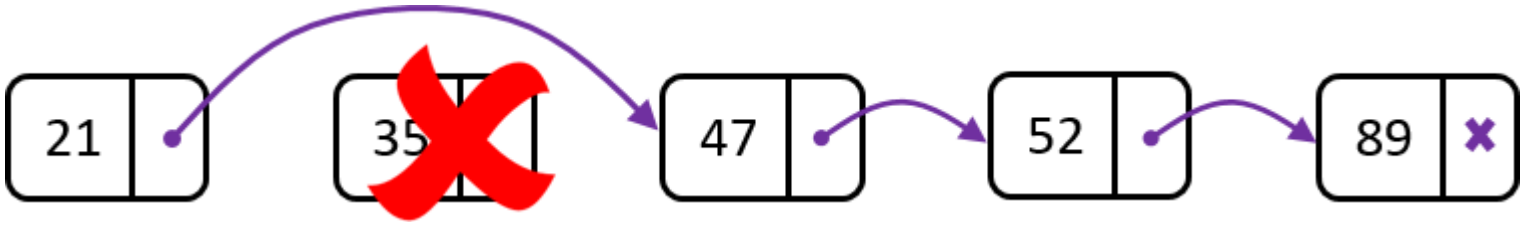
Removing a value from a Singly Linked List

Remove a node in middle of the list

Memory Address	Data Element	Pointer
0	21	1
1	35	2
2	47	4
3	89	null
4	52	3

Removing a value from a Singly Linked List

Remove a node in middle of the list



Memory Address	Data Element	Pointer
0	21	2
~~1~~	~~35~~	~~2~~
2	47	4
3	89	null
4	52	3

Deleting a node

- Steps
  - Find the desirable node
  - Release the memory occupied by the found node
  - Set the pointer of the predecessor of the found node to the successor of the found node
- There are 4 cases,
  - List has only one node
  - Remove first node
  - Remove last node
  - Remove a node from anywhere in the list

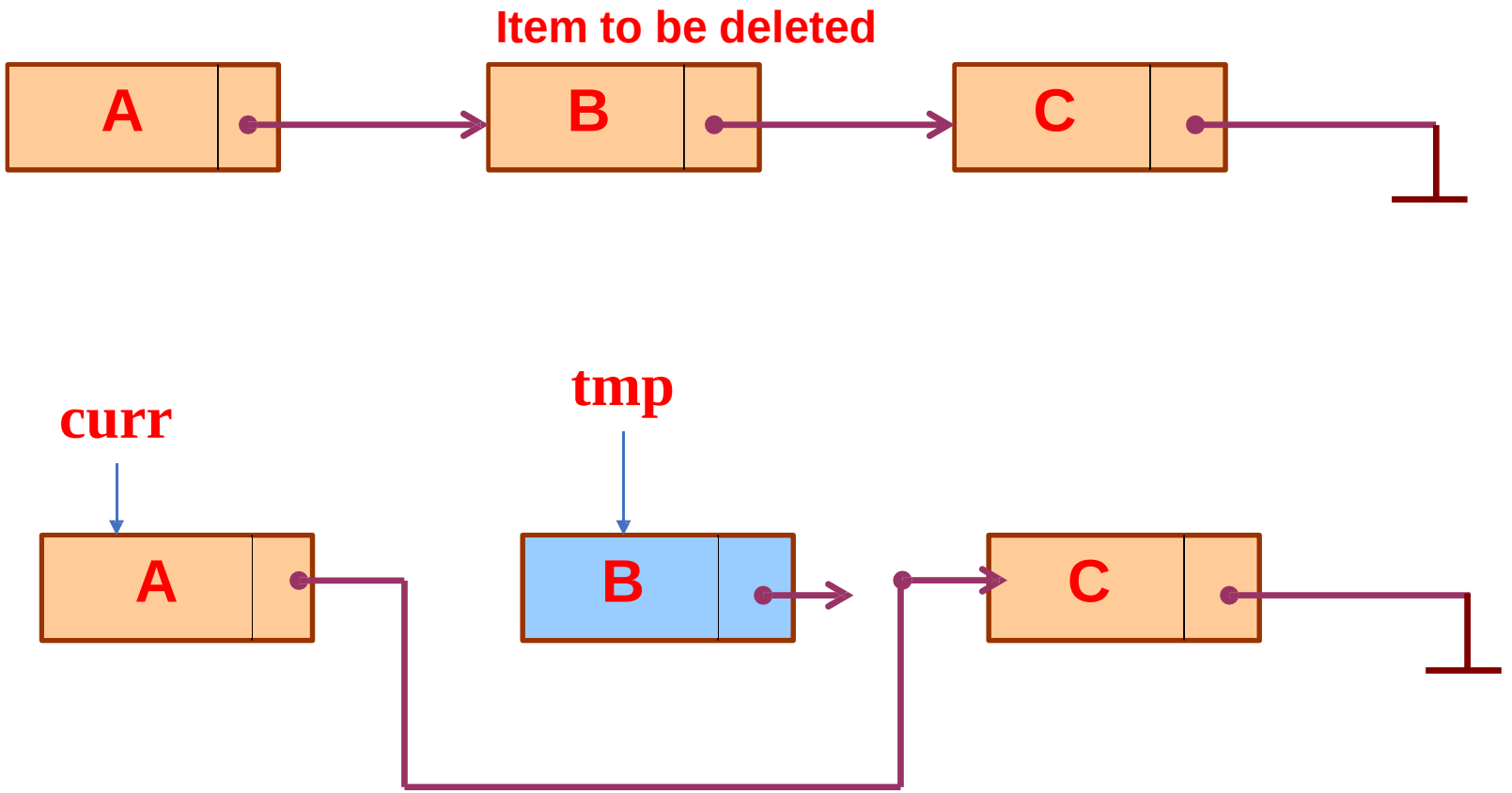
Delete node in a linked list

- **Pseudo Code:**

```
delete (node *head, char d)
node *p, *q;
q = head;
p = head → next;
If(q → data == d )   // start node deletion
{ head = p;
delete (q); }
else { While ( p → data != d) {
p = p→ next;
 q = q → next;}
```

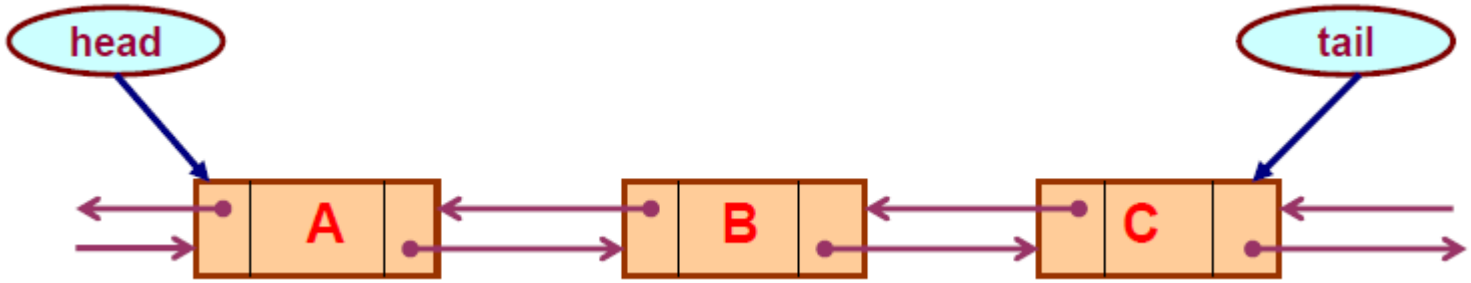
```
If (p → next == NULL)  // Last Node
deletion
{
  q→ next = NULL;
  delete (P)
}
else {
 q→ next = p→ next;   // middle node
 delete (p);
    }
}
}
```

Node Deletion – General Case



Doubly Linked Lists

- A linked list that navigates to both forward and backward directions through the data elements.
- Consists of nodes where each node contains a data field and two references(links), one for the previous node and one for the next node.
- Always the **previous**field of the first node must be NULL.
- Always the **next**field of the last node must be NULL.



Advantages of Doubly Linked Lists

- Can allocate or de-allocate memory easily when required during its execution.
- It is one of most efficient data structure to implement when traversing in both direction is required.

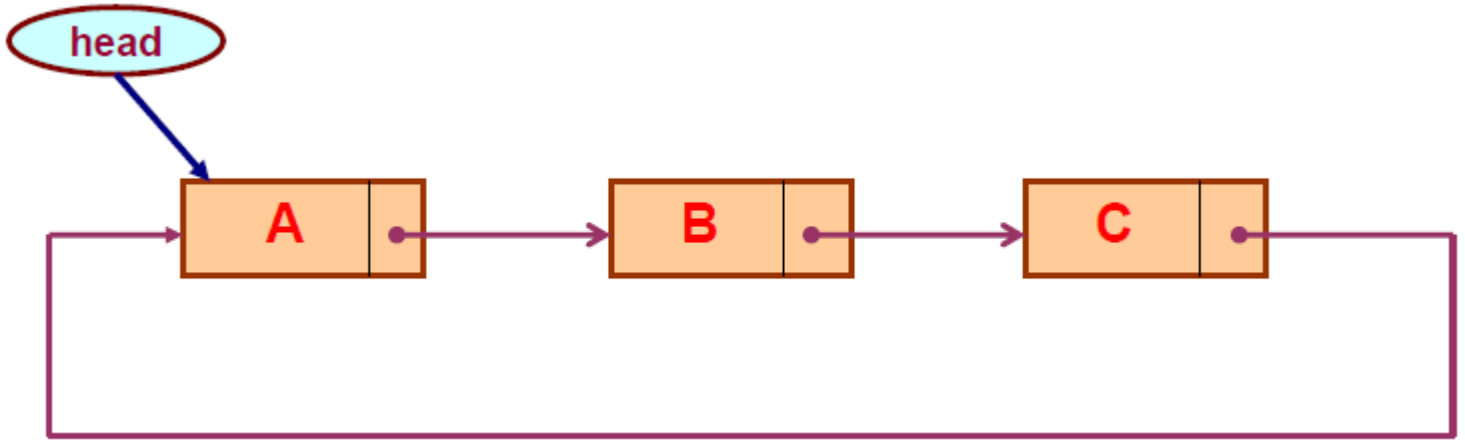
Applications of Doubly Linked Lists

- Used in front and back navigations. (web browsers)
- It is also used to represent various states of a game.
- Used to implement Undo and Redo functionality.

Singly linked List vs Doubly Linked List

Circular Linked Lists

- A circular linked list is used to form a circular chain of data elements.
- The last node of a circular linked list holds the address of the first node.
- Can be either a singly circular linked list or doubly circular linked list.
- There are no start or end node. Hence can be traversed from any node.



Drawbacks of using Arrays for implementation of stacks and queues

- Create array of predefined size and cannot increase the size of the array if there more elements to insert.
- If large array is created, lot of memory will be wasted.

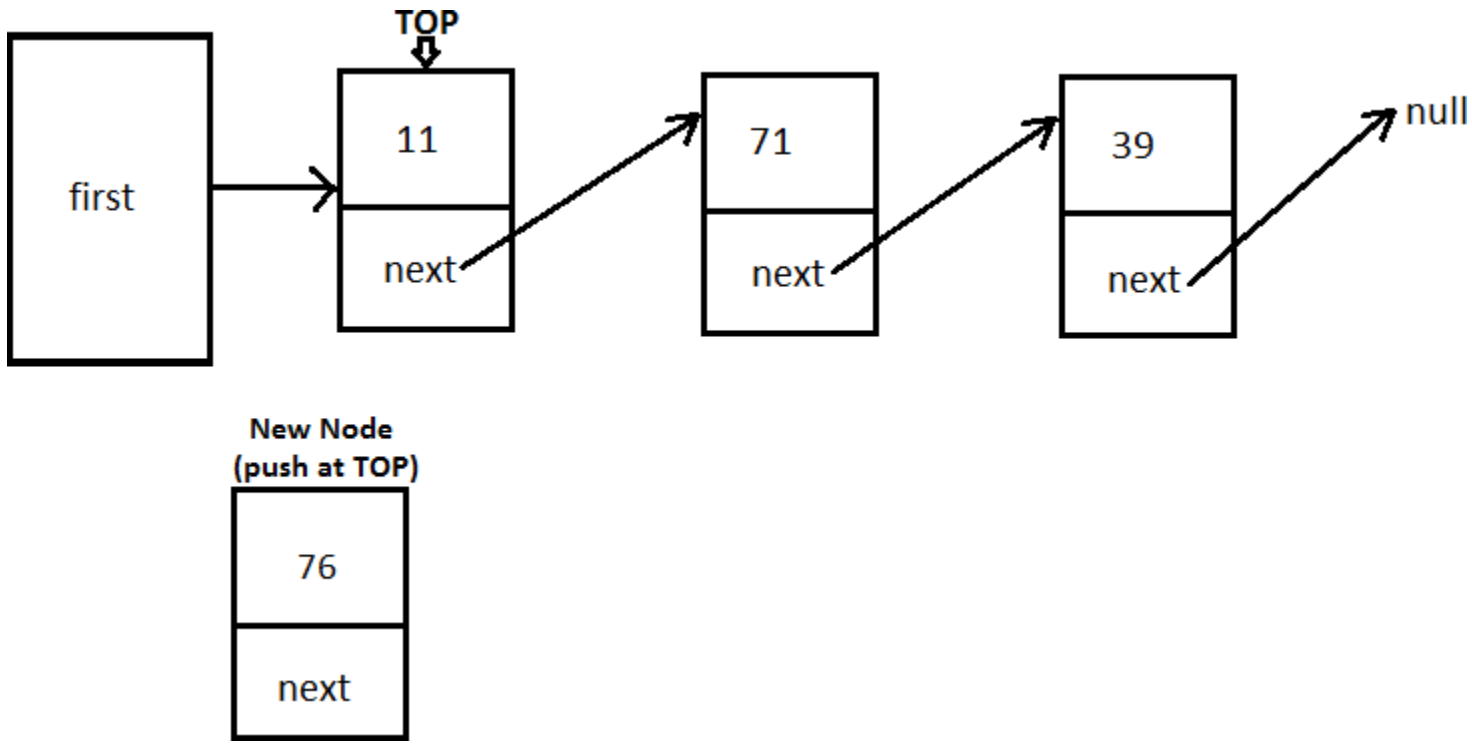
Solution : Linked list implementation

- It dynamically increase or decrease the size of the data structure as per the requirement.

Stacks Implementation using Singly Linked List

- Two ways to decide top element,
- Tail node
- Head node ← preferred

Stacks using Singly Linked List



Stack Operations

- **push() :** Insert the element into linked list which is the top node of Stack.
- **pop() :** Return top element from the Stack and move the top pointer to the second node of linked list or Stack.
- **peek():** Return the top/head element of the linked list.
- **display():** Print all element of linked list by traversing each node in the list.

Adding a node to the stack (Push operation)

1. Create a node first and allocate memory to it.
2. If the list is empty then the item is to be pushed as the start node of the list.
3. If there are some nodes in the list already, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

Deleting a node from the stack (POP operation)

1. Check for the underflow condition: The stack will be empty if the head pointer of the list points to null.
2. Get a temporary pointer and assign it to top element of the list.
3. Assign top node's next link to top (next node address)
4. Remove the link of the temporary node

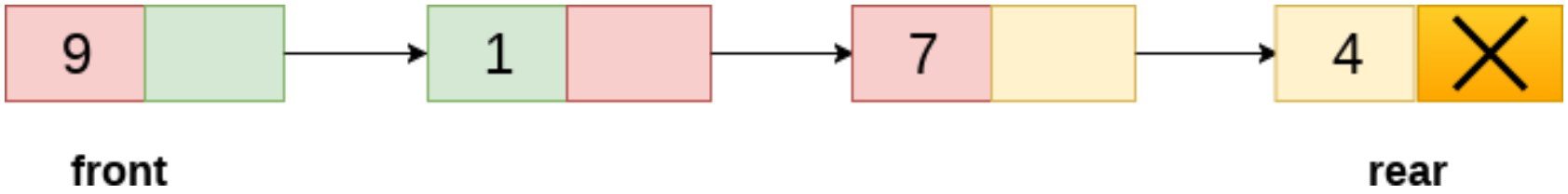
Display the nodes (Traversing)

1. Copy the head pointer into a temporary pointer.
2. Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

Queues Implementation using Singly Linked List

- The front pointer contains the address of the starting element of the queue
- The rear pointer contains the address of the last element of the queue.

Queues using Singly Linked List



Operation on Linked Queue

- **Enqueue()** - Append the queue by adding an element to the end of the queue. The new element will be the last element of the queue.
- **Dequeue()** - Removes the element that is first inserted among all the queue elements.

Adding elements to Queue. (Enqueue Operation)

1. Create a temporary node
2. Assign data to the node.
3. Assign temp node’s next link to Null if list is empty before.
4. Assign rear and front pointers to new node.
5. If list is not empty, assign rear node’s next link to temp or new node and assign rear pointer to temp node

Remove items from the queue. (dequeue operation)

1. Check for the underflow condition: The queue will be empty if the front pointer of the list points to null.
2. Get a temporary pointer and assign it to front element of the list.
3. Assign front node’s next link to front (next node address)
4. Remove the link of the temporary node (free the temp node)

Questions?