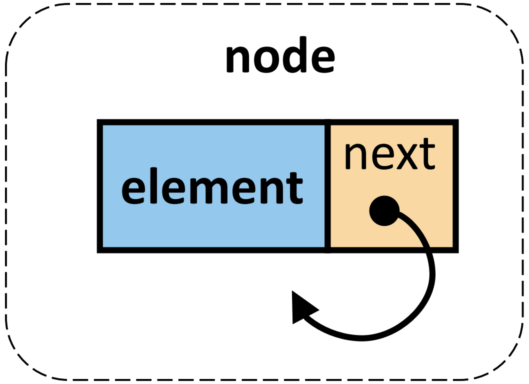
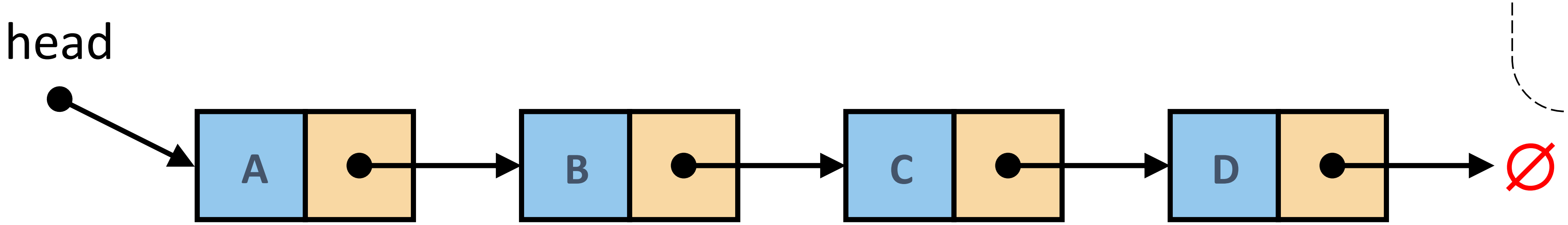
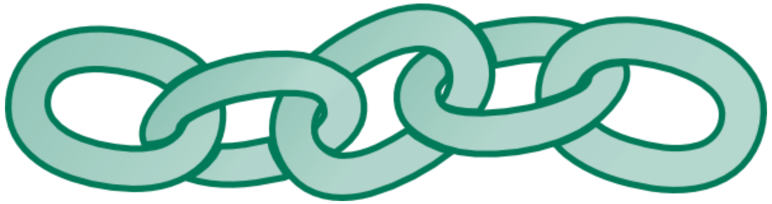
Singly Linked List

Singly Linked List



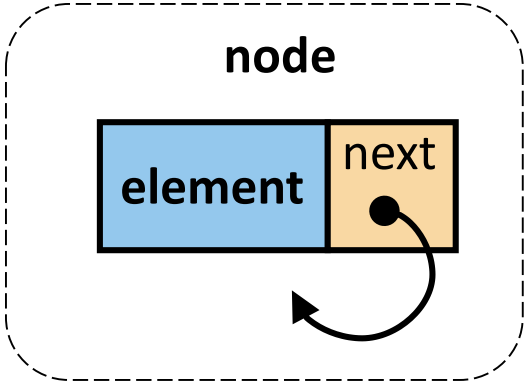


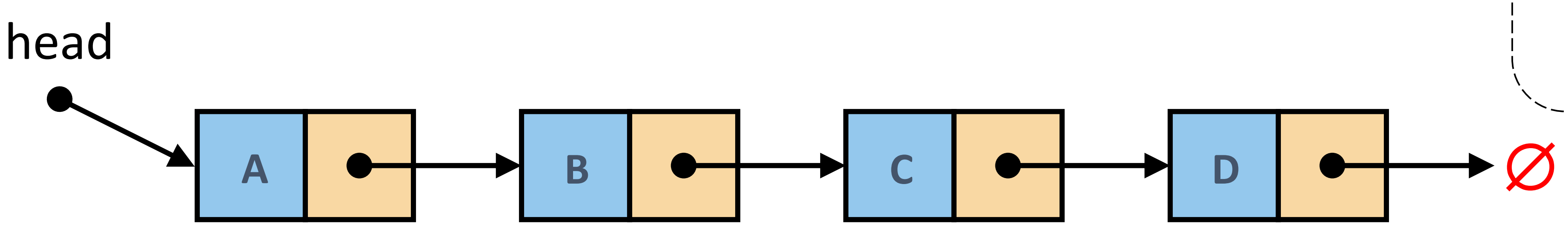
- A singly linked list is a data structure consisting of a sequence of nodes, linked to each other by pointers, starting from a head pointer.
- Each node stores
  - Element -- i.e., the data (same) type of your application
  - Link to the next node



(Singly) Linked List == Chain of Links

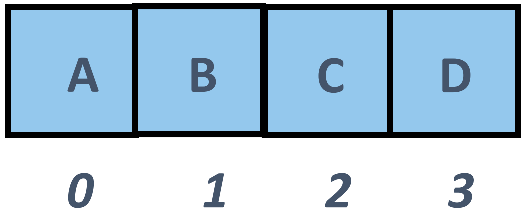
Singly Linked List

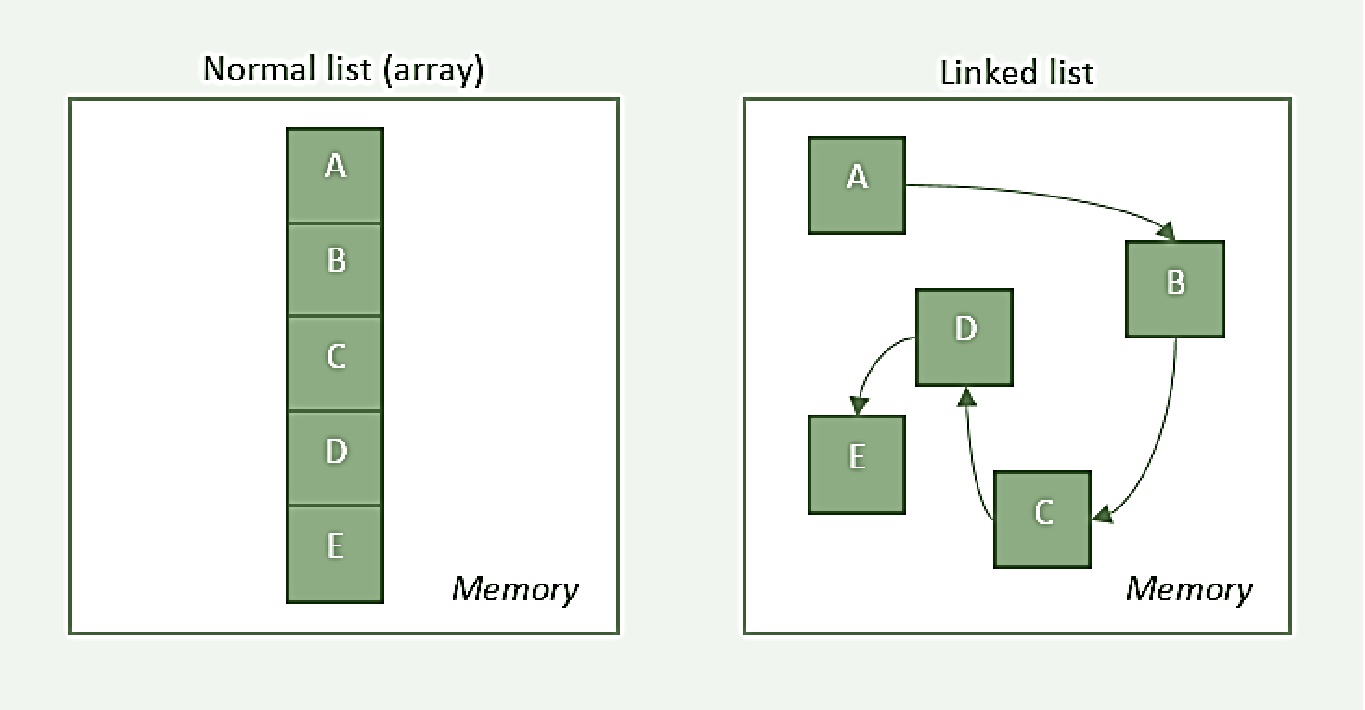




- ✓ Each link in a linked list is an object (also called an element, node, etc.)
- ✓ Each node holds a pointer (a reference, an address) to the location of the next node.
- ✓ The last link in a singly linked list points to null, indicating the end of the list.
- ✓ A linked list can grow and shrink dynamically at run-time (i.e., the time at which your program is running, after it has been compiled), limited only by the amount of physical memory available.

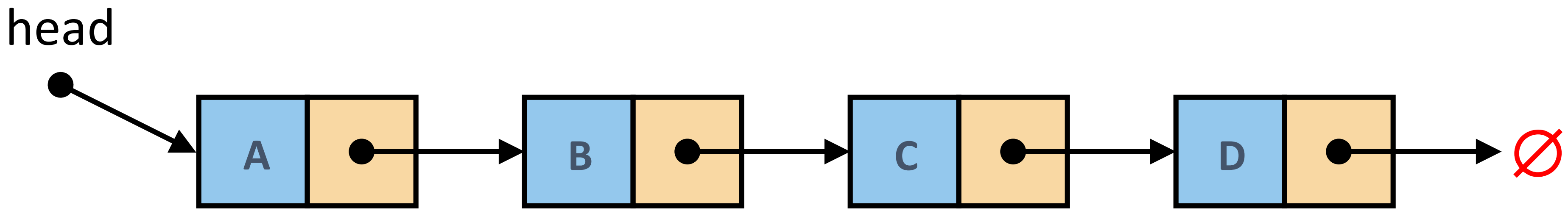
Static (1-D) Array

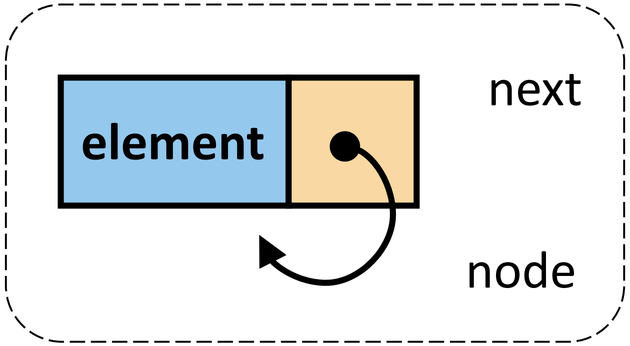




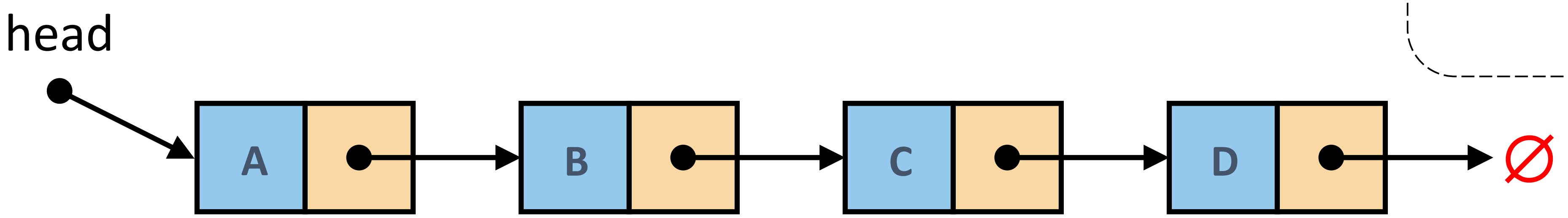
Array elements are stored at consecutive memory addresses
Linked list elements (node) stored anywhere.

Singly Linked List





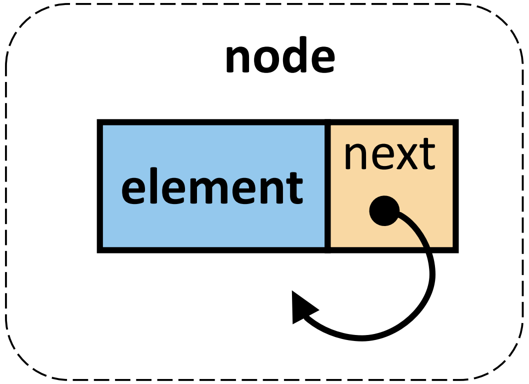
Singly Linked List

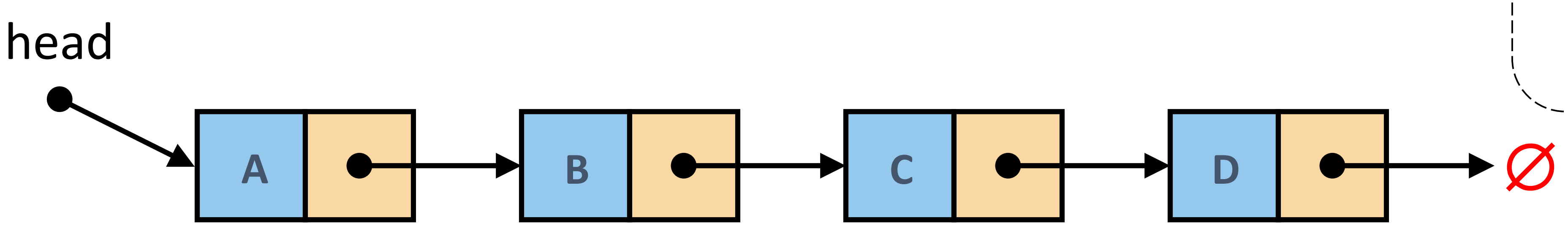


When to Use (Singly) Linked Lists?

- You need to do constant insertions and deletions.
- You are not sure how many items will be in the list (i.e., dynamic data).

Singly Linked List





PROS

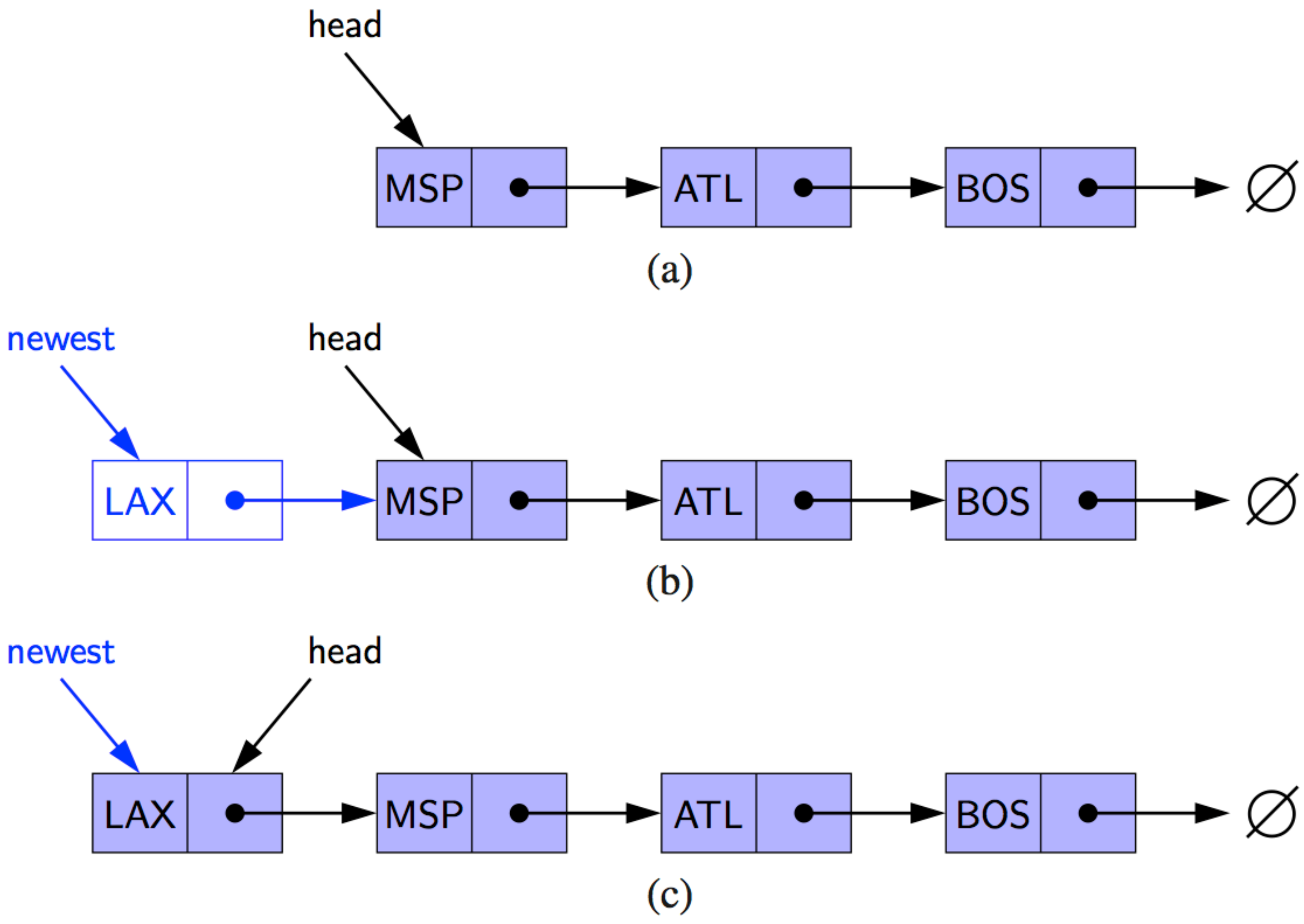
- **Insertions and deletions are quick.**
- **Grows and shrinks as needed.**

CONS

- **Random access is slow.** Nodes in a linked list must be accessed <u>sequentially</u> (i.e., it can be slow to access a specific object).
- **Memory is a concern.** Each object in a singly linked list requires data (i.e., element) as well as one pointer (i.e., reference) to other nodes in the singly linked list. 1-D arrays use significantly less memory, each entry [i] in an array only requires memory to store its data.

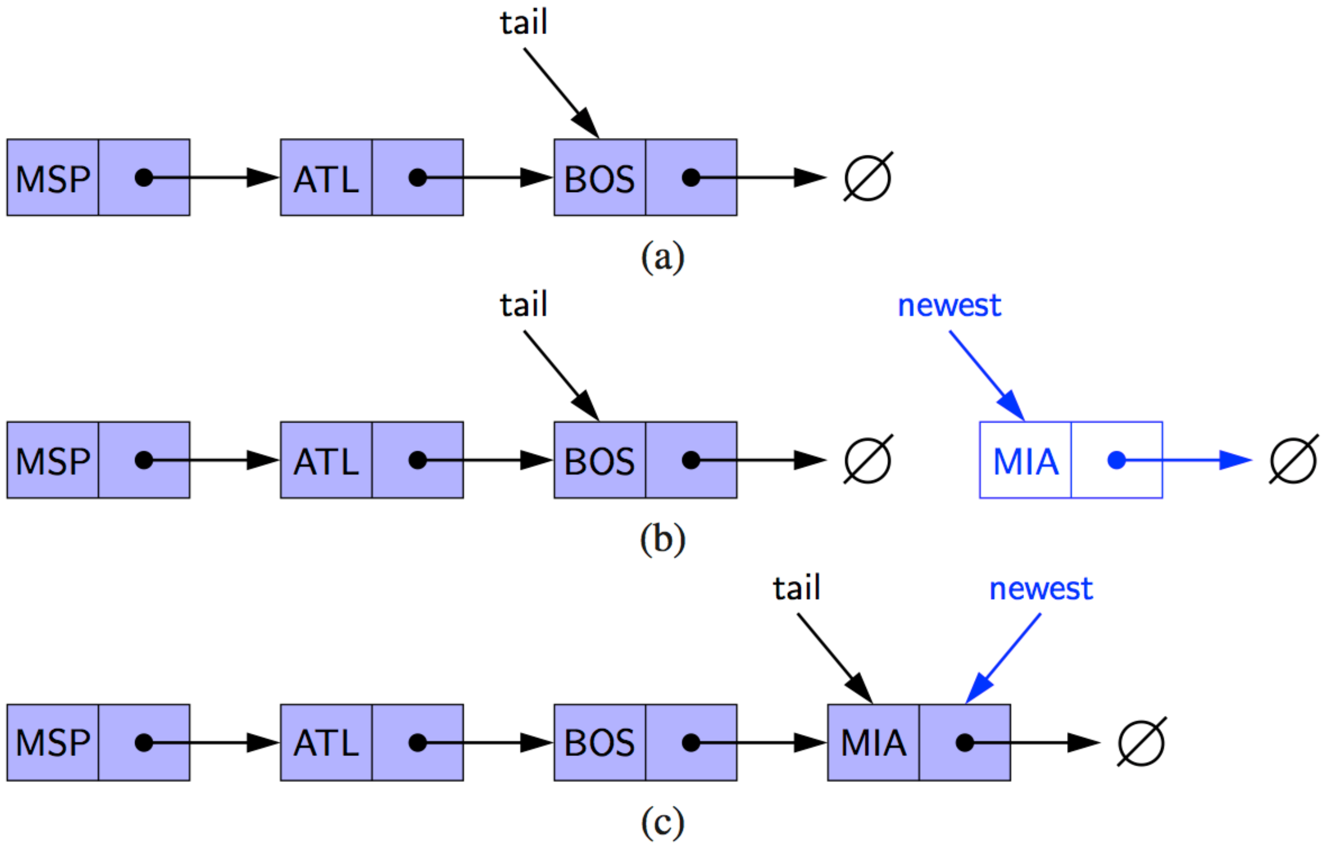
Inserting at the Head

- Allocate new node
- Insert new element
- Have new node point to old head
- Update head to point to new node



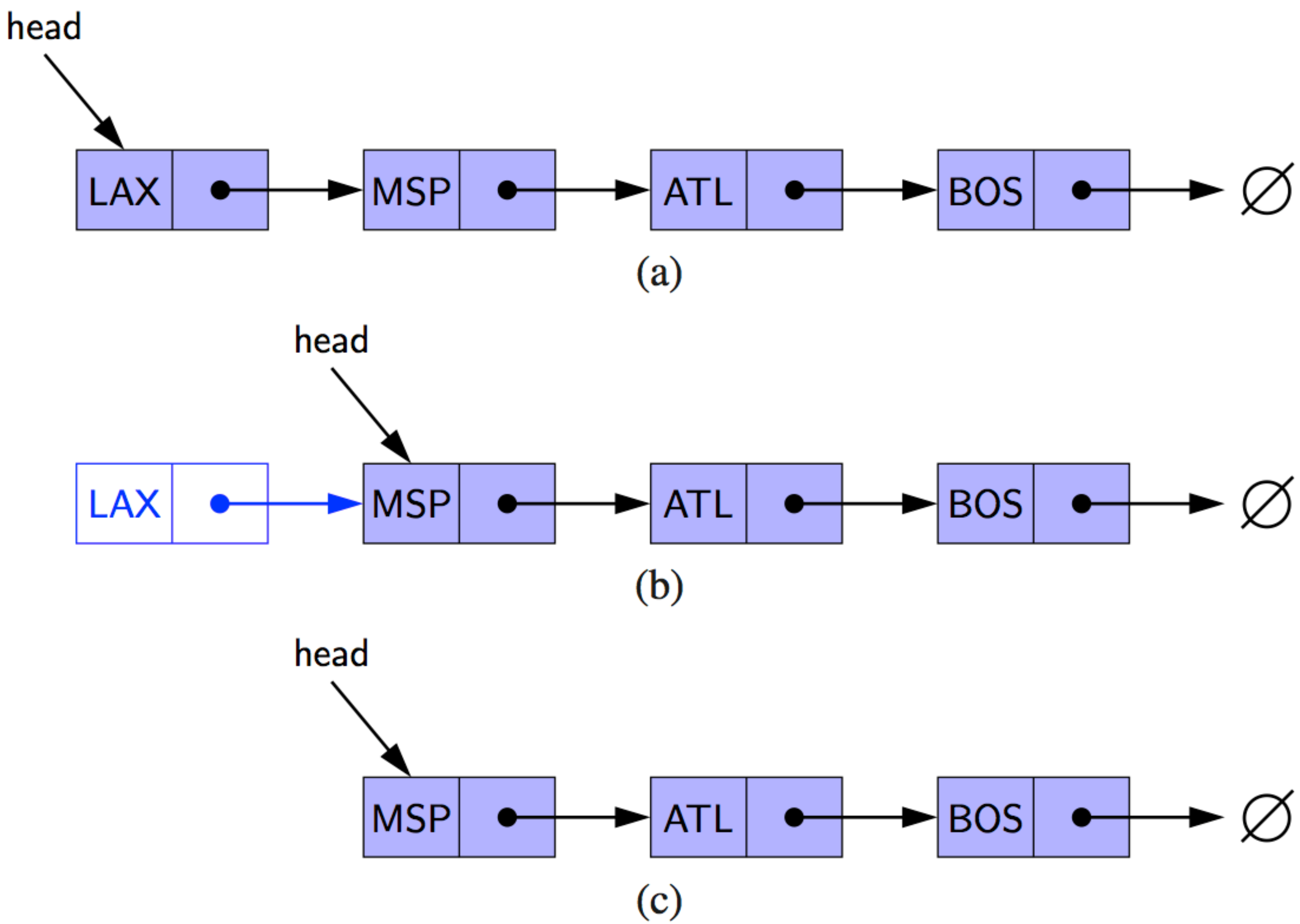
Inserting at the Tail

- Allocate a new node
- Insert new element
- Have new node point to null
- Have old last node point to new node
- Update tail to point to new node



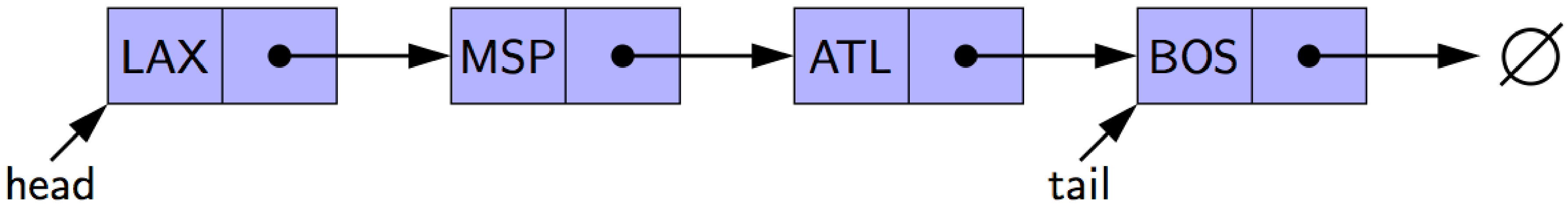
Removing at the Head

- Update head to point to next node in the list
- Allow garbage collector to reclaim the former first node



Removing at the Tail

- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node
- We need an auxiliary pointer (reference) to traverse the list from head until the node previous to the tail node

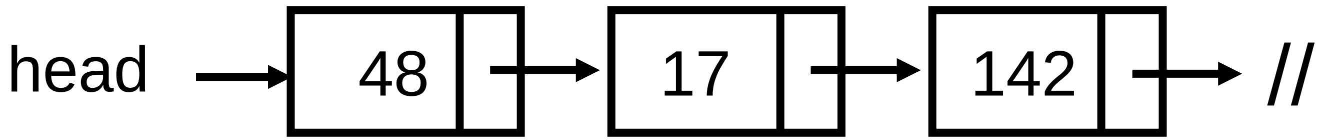


Singly Linked List Traversal

- **Traversal means “visiting” or examining each node.**
- **Singly linked list**
  - **Start at the beginning**
  - **Go one node at a time until the end**

Insertion into Singly Linked Lists

- **You have a singly linked list**
  - **Perhaps empty, perhaps not**
  - **Perhaps ordered, perhaps not**



- **You want to add an element into the singly linked list**

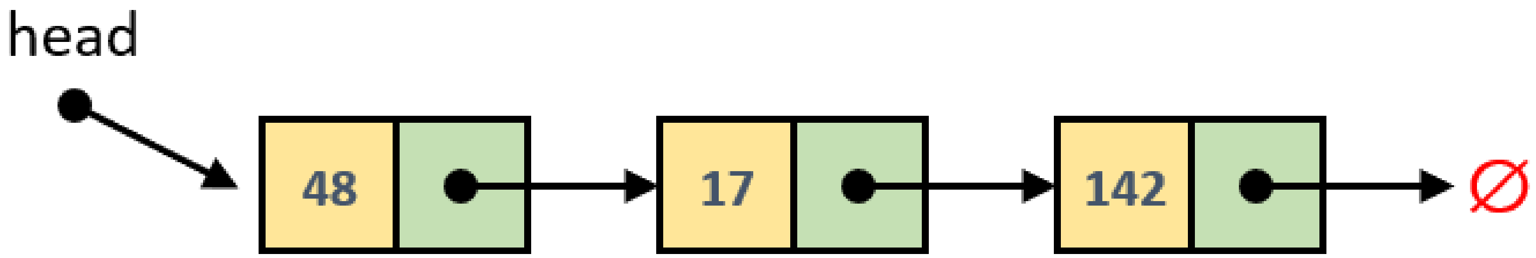
Adding an Element to a Singly Linked List

Involves two steps:

- **Finding the correct location**
- **Doing the work to add the node**

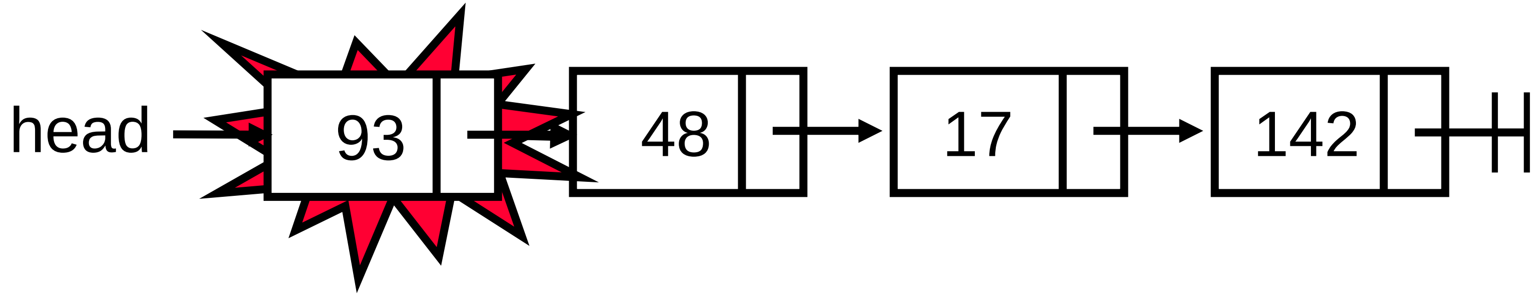
Finding the Correct Location

- **Three possible positions:**
  - **The front**
  - **The end**
  - **Somewhere in the middle**



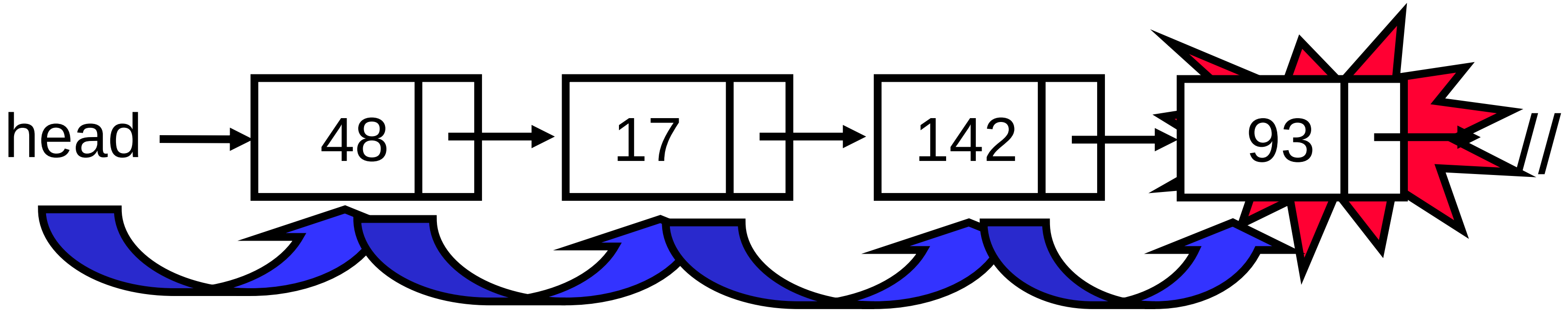
Inserting to the Front

- **There is no work to find the correct location**
- **Empty or not, head will point to the right location**



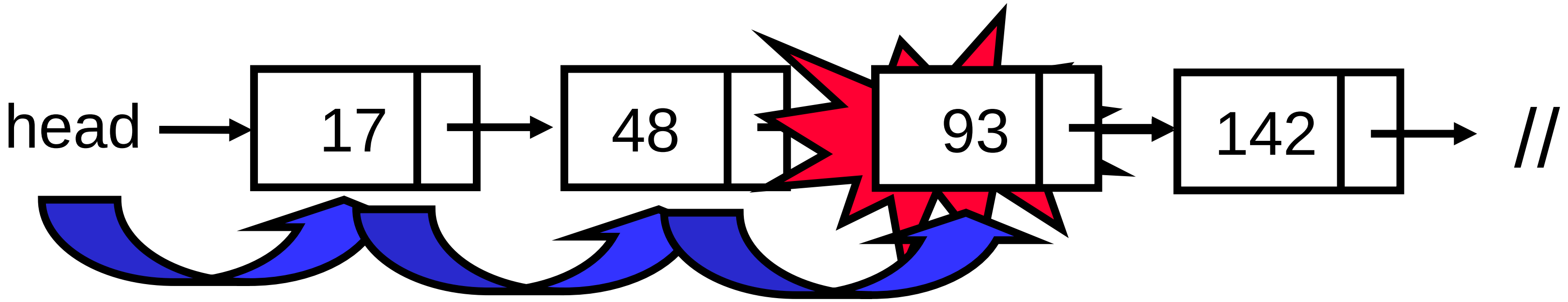
Inserting to the End

- **Find the end of the list (when at NULL)**
  - **Recursion or iteration**



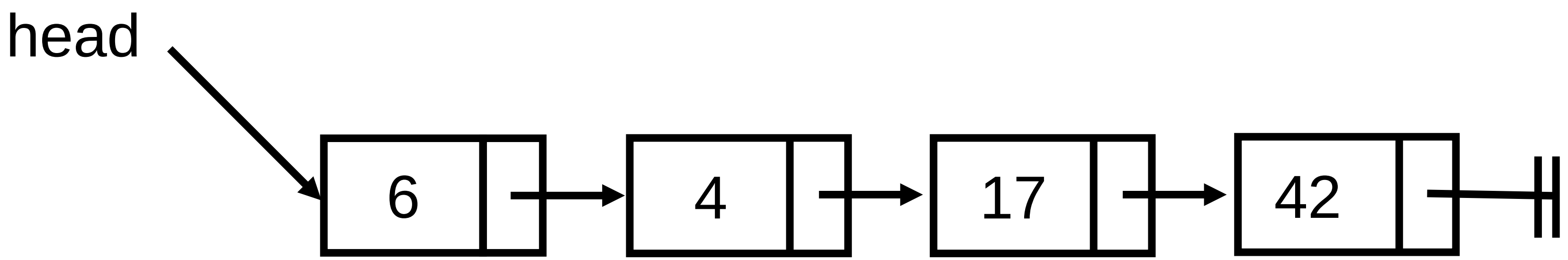
Inserting to the Middle

- **Used when order is important**
- **Go to the node that should follow the one to add**
  - **Recursion or iteration**



Deleting an Element from a Singly Linked List

- **Begin with an existing singly linked list**
  - **Could be empty or not**
  - **Could be ordered or not**



The Scenario

- **Begin with an existing singly linked list**
  - **Could be empty or not**
  - **Could be ordered or not**