Data structures and algorithms
Spring 2025

LIST

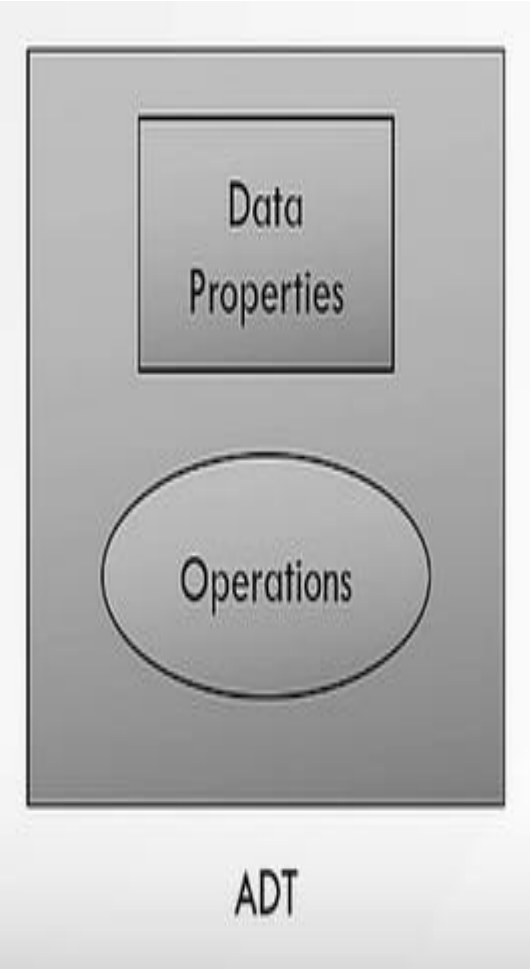
Lecturer: Do Thuy Duong

Contents

- Abstract Data Type (ADT)
- List ADT
- List ADT implementation
  - Array-based list
  - Linked list
    - Singly linked list
    - Circular linked list
    - Doubly linked list

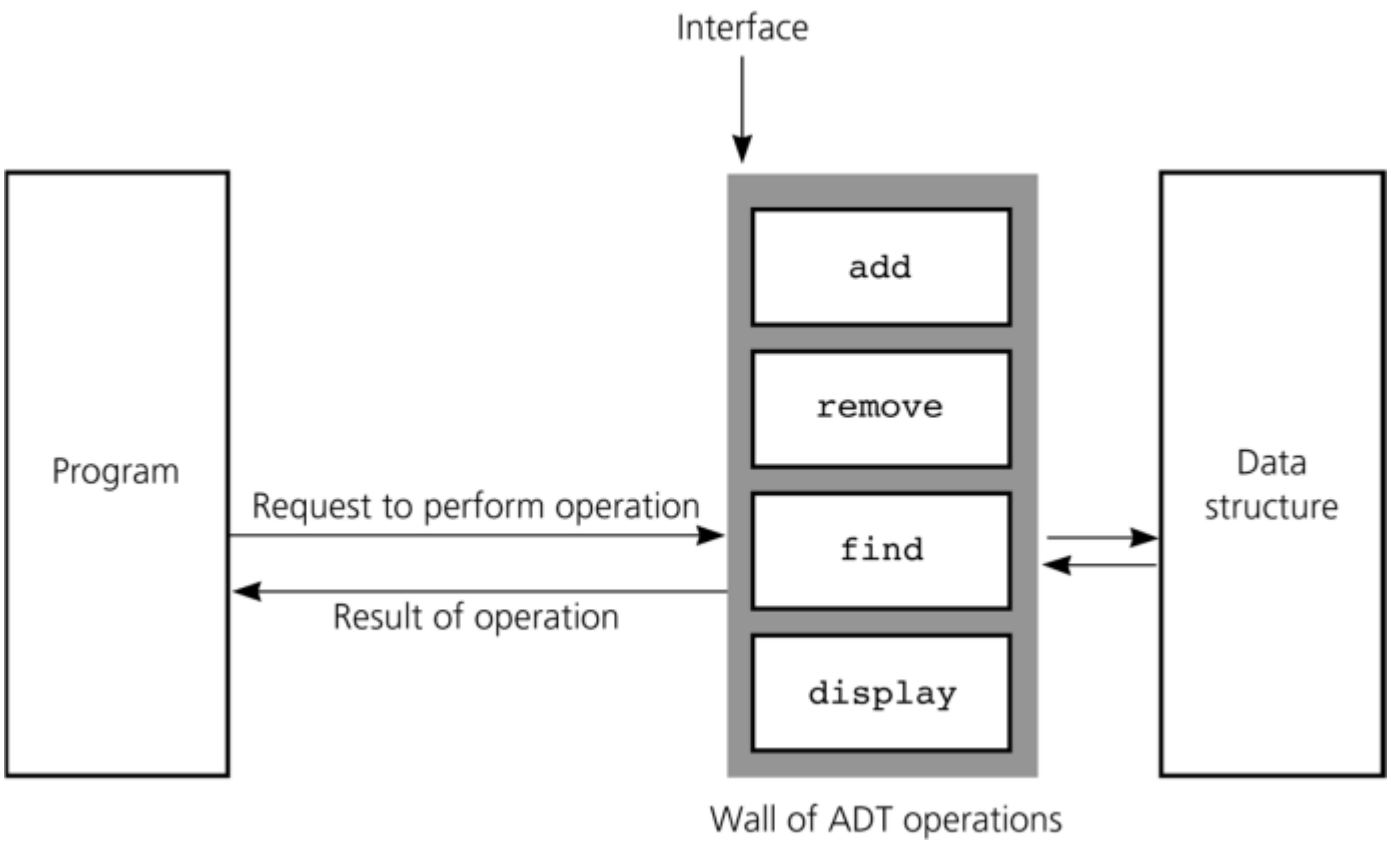
Abstract Data Type (ADT)

- **An ADT is a model of a data type:**
  - **Data properties**
  - A list of **operations** on that data (What operations an ADT can have, not how to implement them)
- Example: The **List** ADT:
  - Group of elements
  - Operations: Add items, Remove items, Find items, Display items…



Abstract Data Type (ADT)

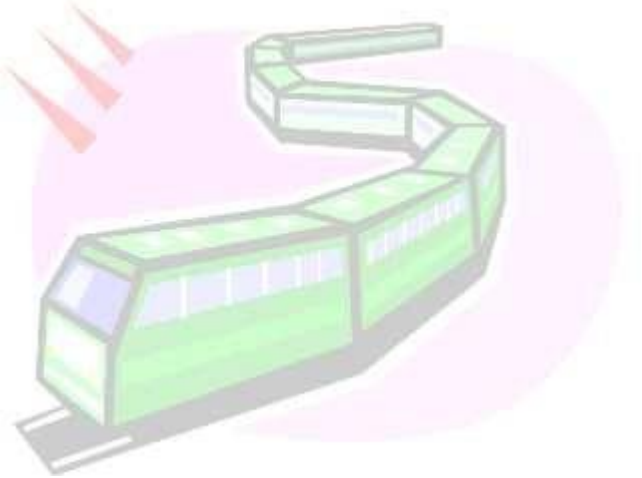
- **We use A Data Structure to implement an ADT.**



Abstract Data Type (ADT)

- **We use A Data Structure to implement an ADT.**

Abstract data types	Data structures
List	Array-based List Linked list
Stack	Array based stack Linked list based stack
Queue	Array based queue Linked list based queue
Map	Tree map Hash map/ Hash table
Vehicle	Bicycle Car Truck



List

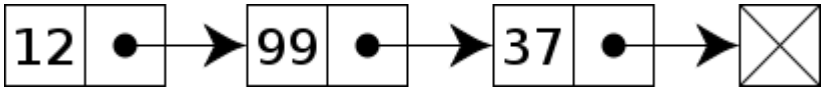
What **it is** (conceptual)?

When **we use it** (applications)?

How **we implement it** (implementation)?

List ADT

- **Definition**



  - A sequence of **zero or more elements** of the same type.
  - Except for the first and last items, each item has a unique predecessor and a unique successor.
  - First item (**head** or **front**) do not have a predecessor.
  - Last item (**tail** or **end**) do not have a successor.
  - Items are referenced by their position within the list

List ADT

- **List Operations**
  - Create an empty list.
  - Determine whether a list is empty.
  - Determine the number of items in a list.
  - Add an item at a given position in the list.
  - Remove the item at a given position in the list.
  - Remove all the items from the list.
  - Retrieve (get) item at a given position in the list.

List ADT

- UML model

List ADT
+ isEmpty(): boolean + getLength(): int + add(int: pos, ItemType: newItem): void + remove(int: pos): void + removeAll(): void + get(int: Pos): ItemType …

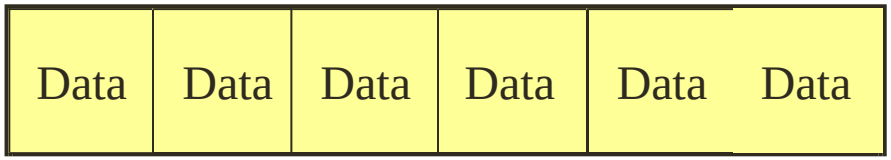
List implementations

- **ADT implementation**
  - **Choose a data structure to represent the ADT's data**.
  - How to choose? Depends on
    - Details of the ADT's operations.
    - Context in which the operations will be used.

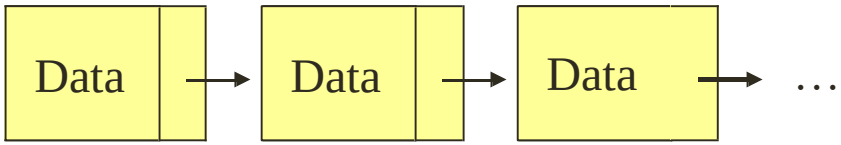
List implementations

A list can be implemented as:

- An array (statically allocated)

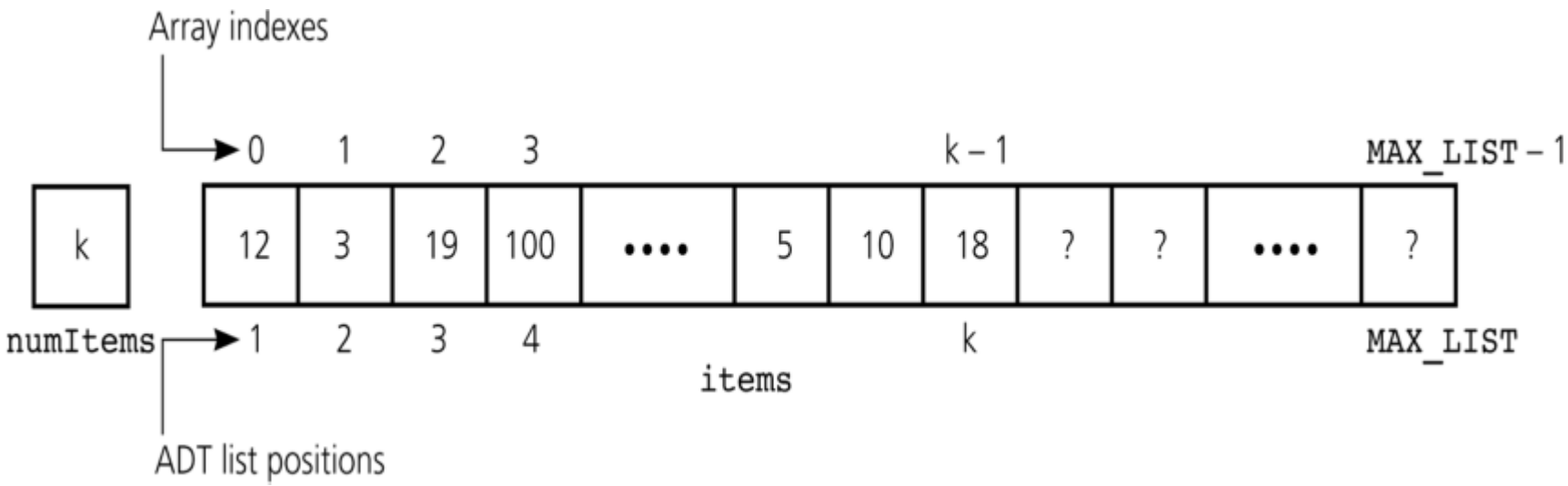


- A linked list (dynamically allocated)



Array-Based List [1]

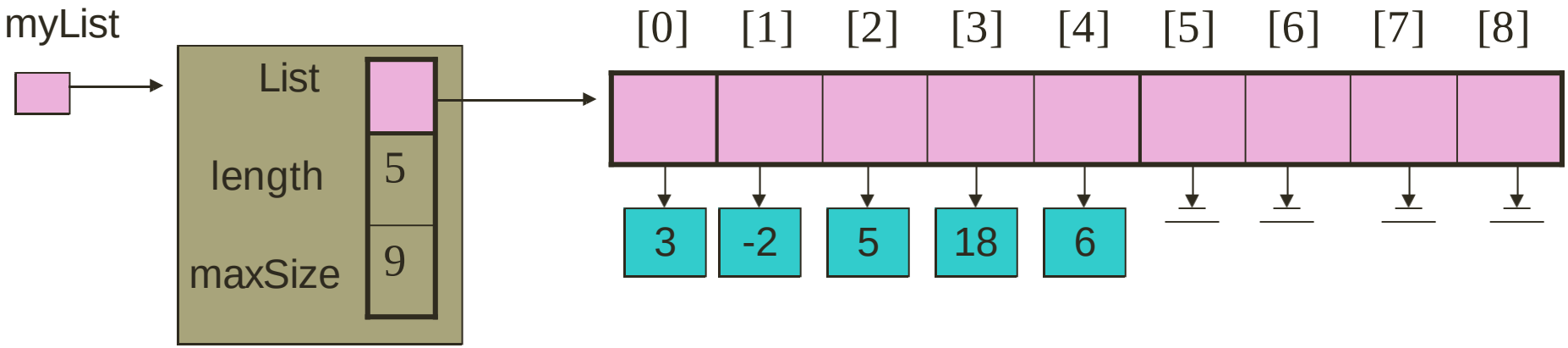
- A list's items are stored in an **array items**
- Both an array and a list identify their items by **index number**.
- A list's **k_{th}** item will be stored in **items[k-1]**
- **length** and **maxSize** are used to indicate the current length and maximum length of the list



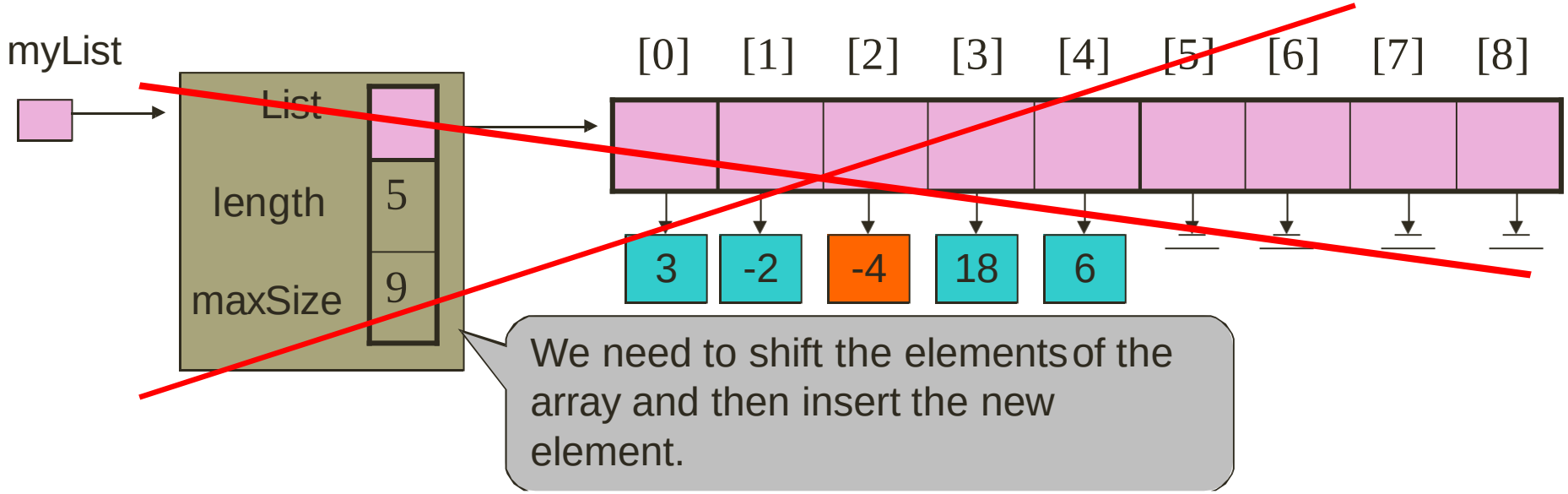
Array-Based List [2]

- isEmpty():
  - The list is empty when length equals to 0
- getLength():
  - Return length
- get(int Pos):
  - Return indexs[Pos-1]
- removeAll():
  - Assign length to 0
- **add() and remove()** ???

Array-Based List [3]

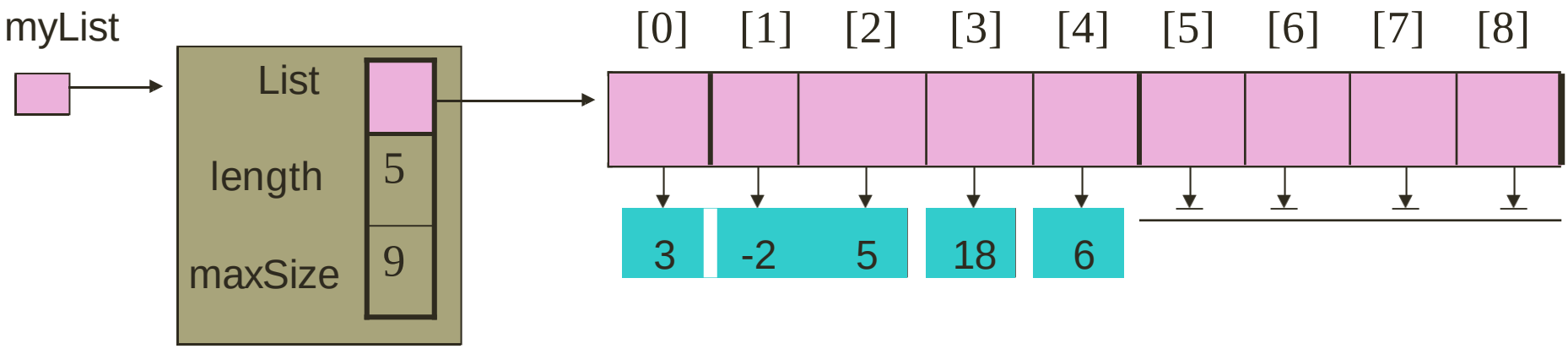


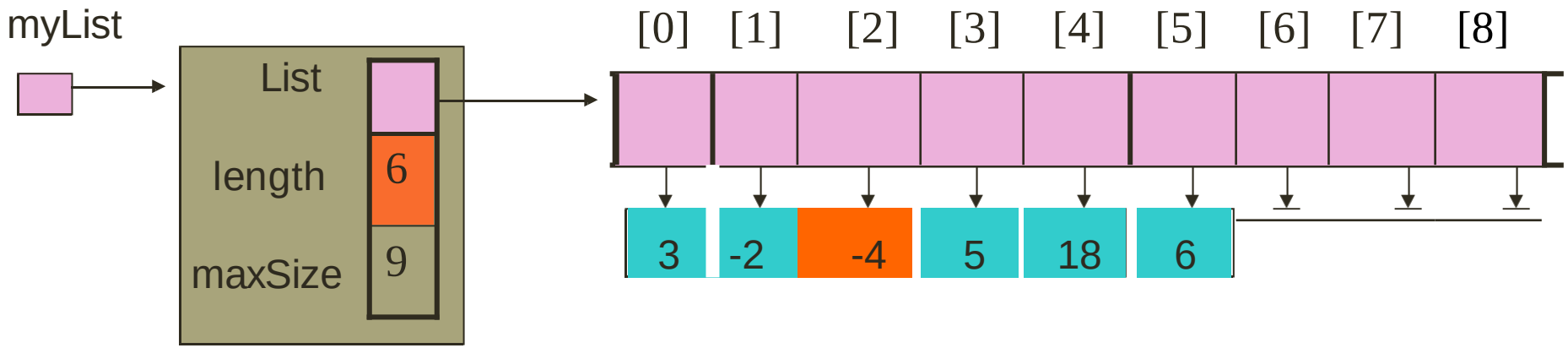

```
add(3,-4);
```



Array-Based List [3]

```
add(3,-4);
```



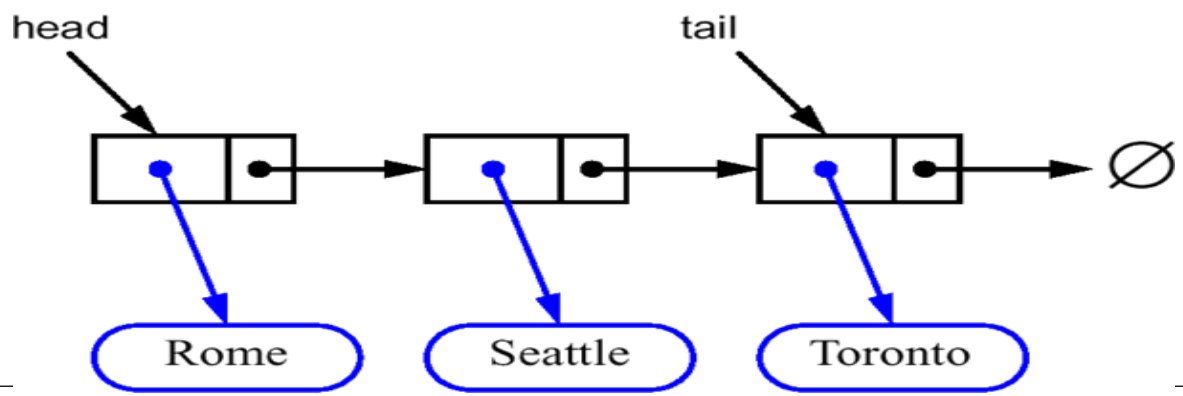


Array-Based List [4]

- **add(), remove()**
  - Time complexity for these operations is O(n), where n is the current length of the list.
- **Advantages**
  - Easy to implement
  - Quick search for a node
- **Disadvantages**
  - Not the ideal solution for problems involving a lot of insertions or deletions (data movement)
  - Size is fixed at runtime

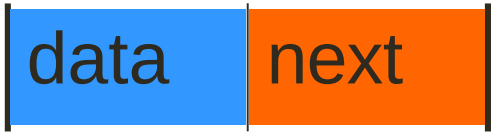
Linked List

- Also called **reference-based list**
- Differences with array-based list:
  - Data storage
    - Arrays store elements in consecutive memory blocks
    - Linked lists store elements in components called nodes, not necessarily in consecutive memory blocks
  - Dynamic size
    - Linked lists can grow at runtime



Singly Linked List[1]

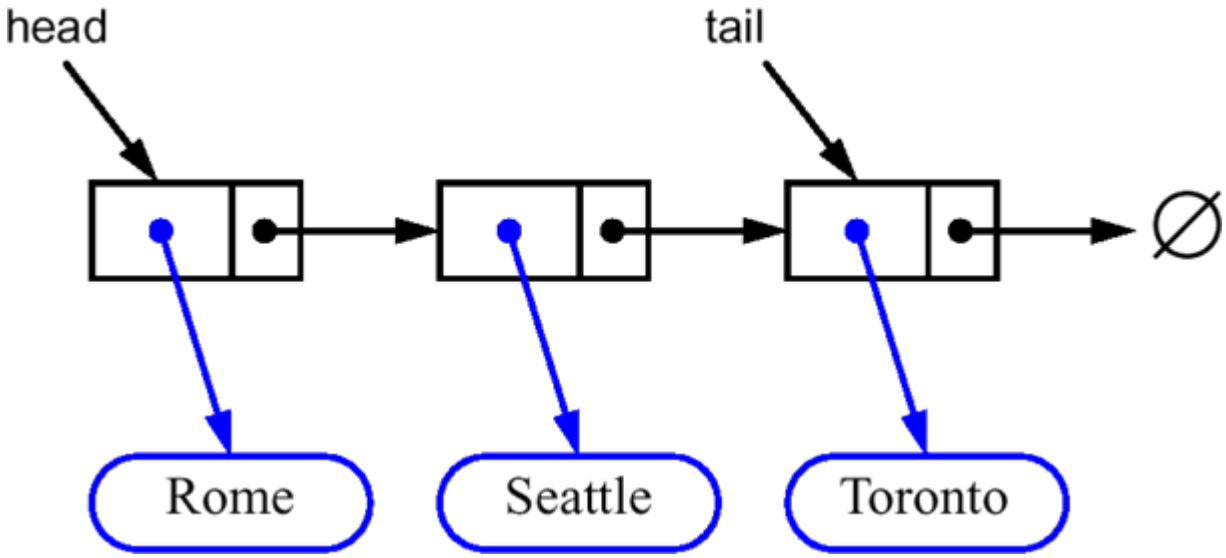
- A list of **nodes**.
- Every node (except the last one) contains the **link** to the next node.
- Every node has two components:
  - data
  - next



- **data** can be a primitive data type or a reference to an object.
- **next** is a reference variable (in Java) or a pointer (in C).

Singly Linked List[2]

- Nodes *(**data**, **next**)* connected in a chain by links



- The **head** or the **tail variables** point to the first and the last node of the list.

Singly Linked List[3]

SLList

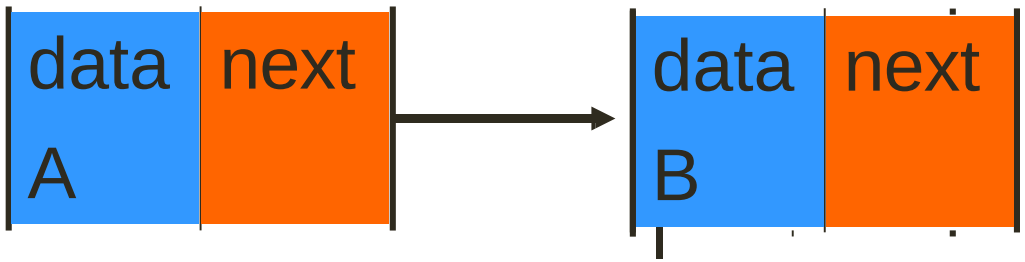
head: SLNode

+ isEmpty(): boolean
+ getLength(): int
+ add(SLNode: node): void
+ addAt(int: pos, SLNode: node): void
+ remove(int: pos): void
+ removeAll(): void
+ get(int: pos): SLNode

SLNode

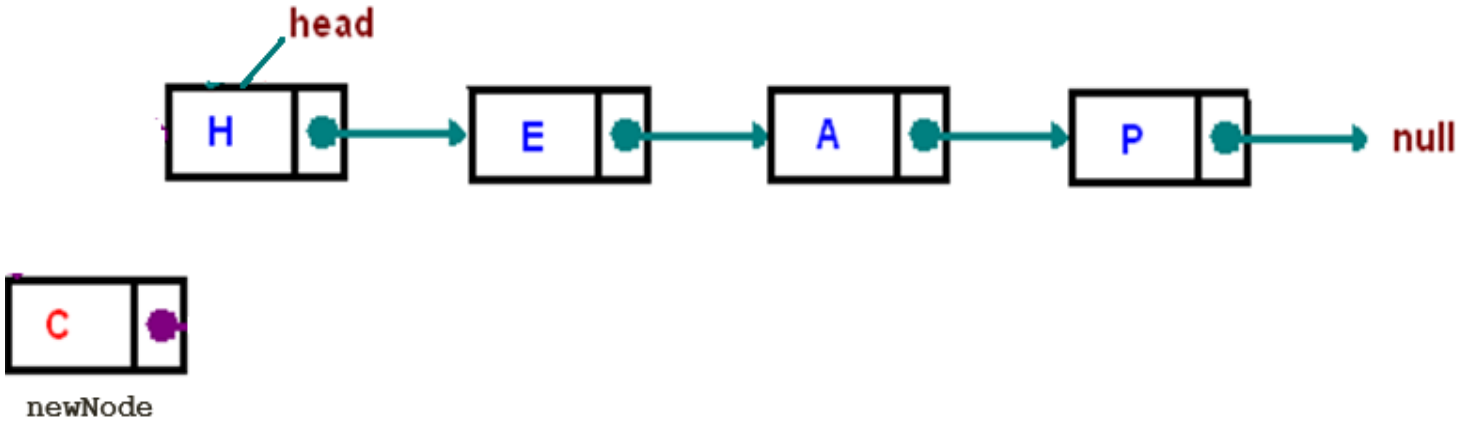
data: DataType
next: SLNode

+setNext(SLNode: node): void
+getNext(): SLNode
+getData(): DataType
+setData(DataType: data):void



Singly Linked List[4]

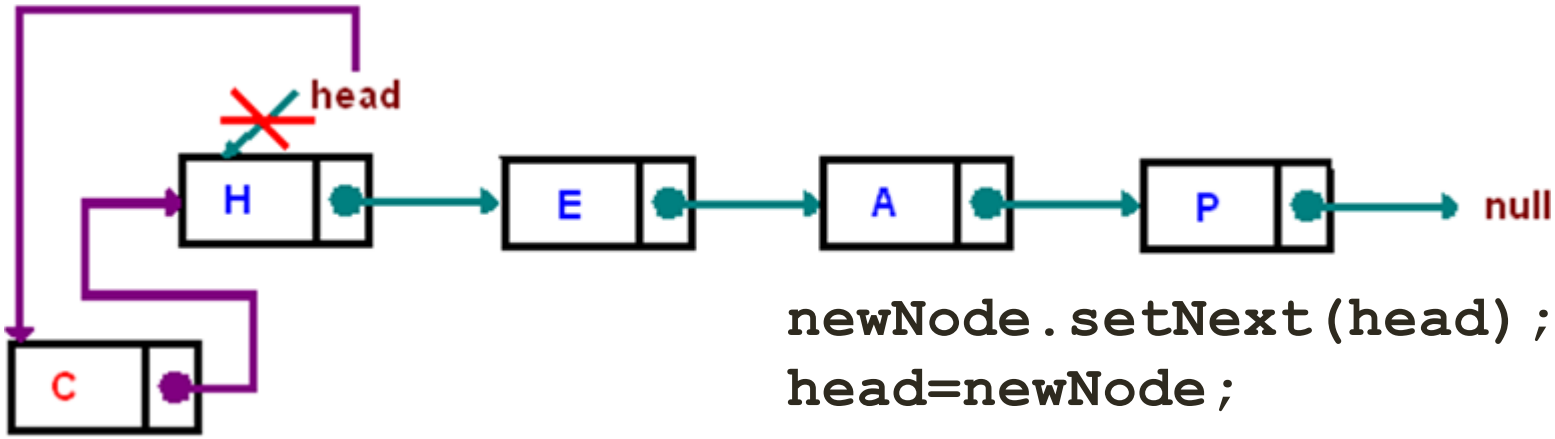
- SLNode:
  - setNext(), getNext(), getData(), setData()
- SLList:
- isEmpty():
  - The list is empty when head equals to null.
- add(SLNode newNode):




```
newNode.setNext(head);
head=newNode;
```

Singly Linked List[4]

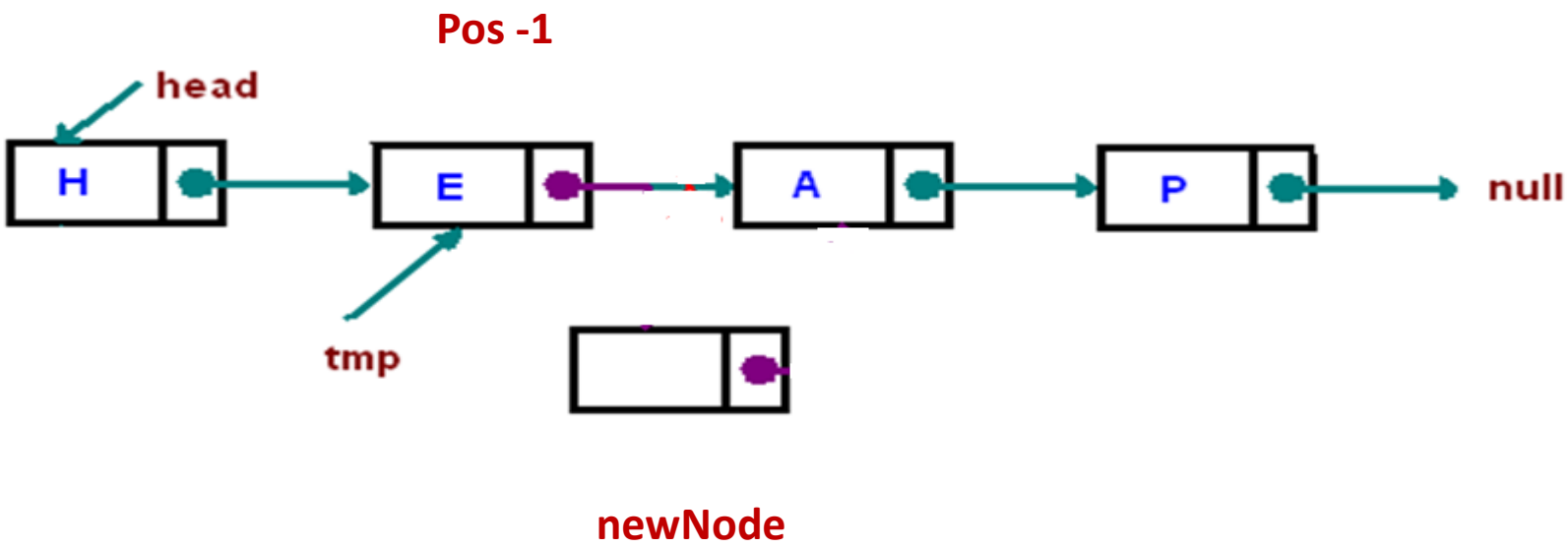
- SLNode:
  - setNext(), getNext(), getData(), setData()
- SLList:
- isEmpty():
  - The list is empty when head equals to null.
- add(SLNode newNode):
  - Insert a node at the beginning of a linked list.




```
newNode.setNext(head);
head=newNode;
```

Singly Linked List[5]

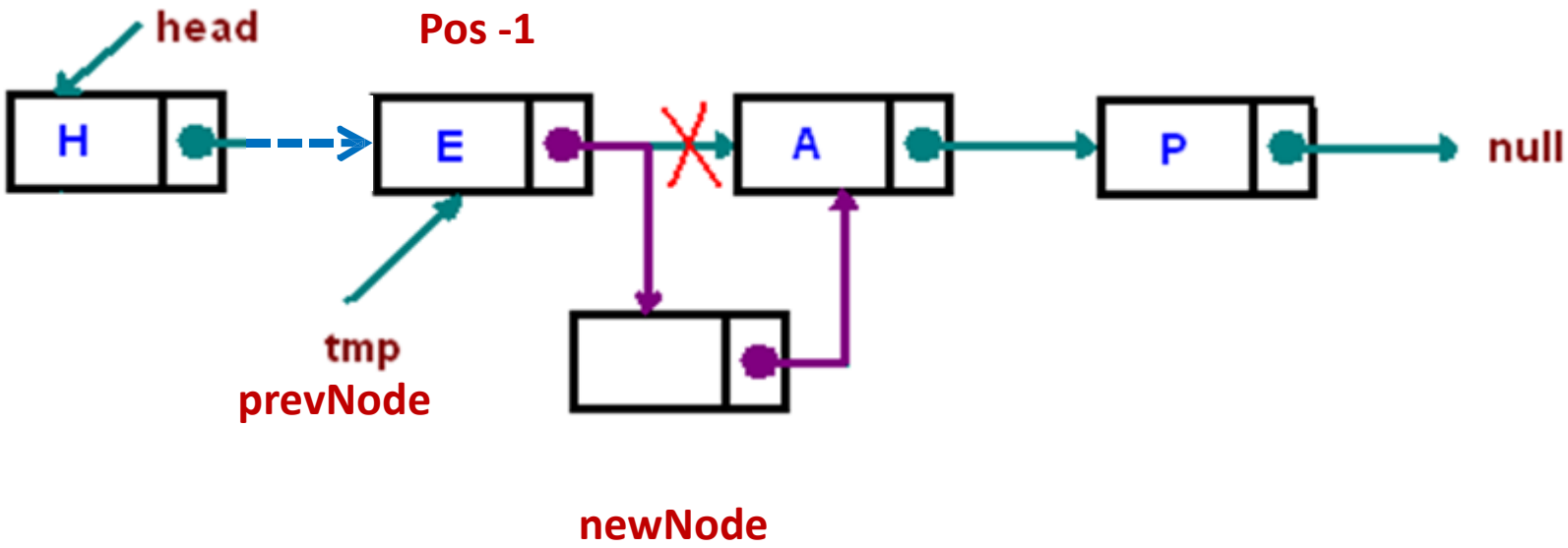
- addAt(int pos, SLNode newNode):
  - Insert a new node at the pos position of a linked list:
    - Travel to the pos-1 position
    - Insert the new node




```
prevNode=traversing(pos-1);
newNode.setNext(prevNode.getNext());
prevNode.setNext(newNode);
```

Singly Linked List[5]

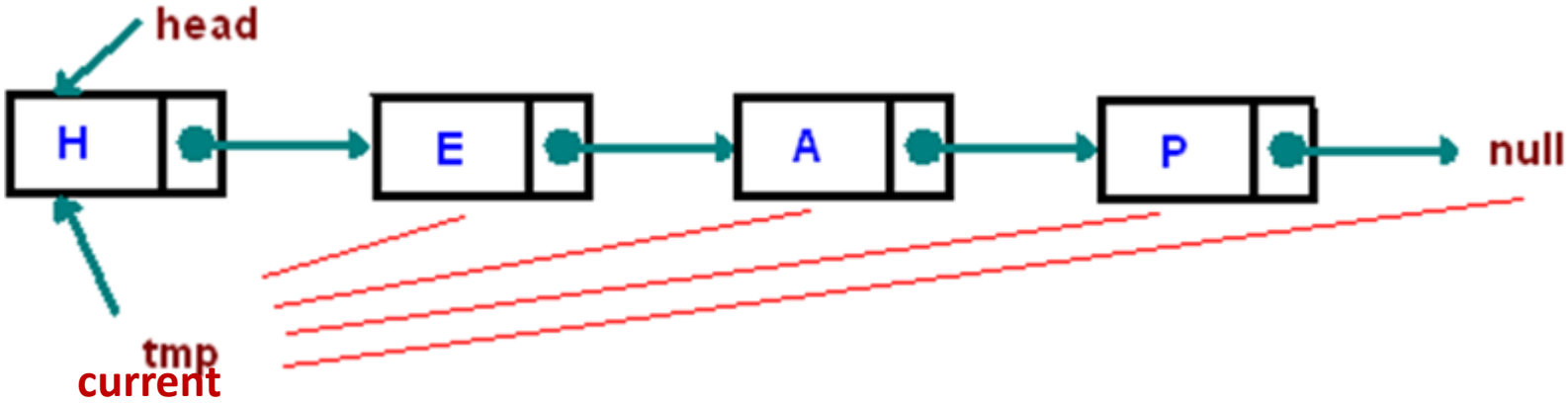
- addAt(int pos, SLNode newNode):
  - Insert a new node at the pos position of a linked list:
    - Travel to the pos-1 position
    - Insert the new node




```
prevNode=traversing(pos-1);
newNode.setNext(prevNode.getNext());
prevNode.setNext(newNode);
```

Singly Linked List[6]

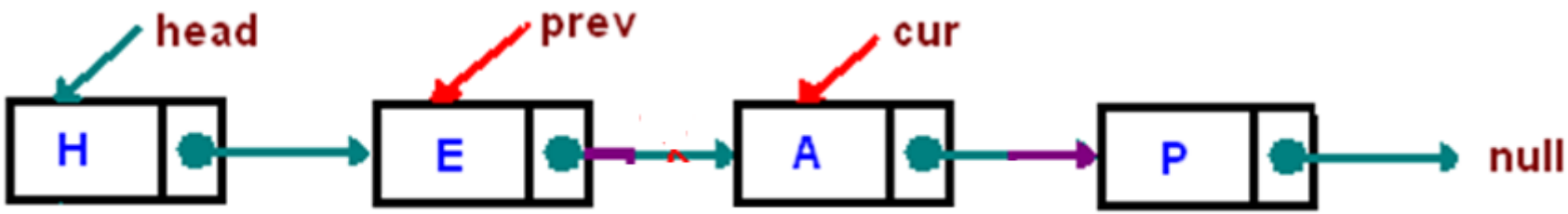
- How to travel to the pos position?




```
int count=1;
SLNode current = this.head;
while (count < pos)
{
      count++;
      current=current.getNext();
}
return current;
```

Singly Linked List[7]

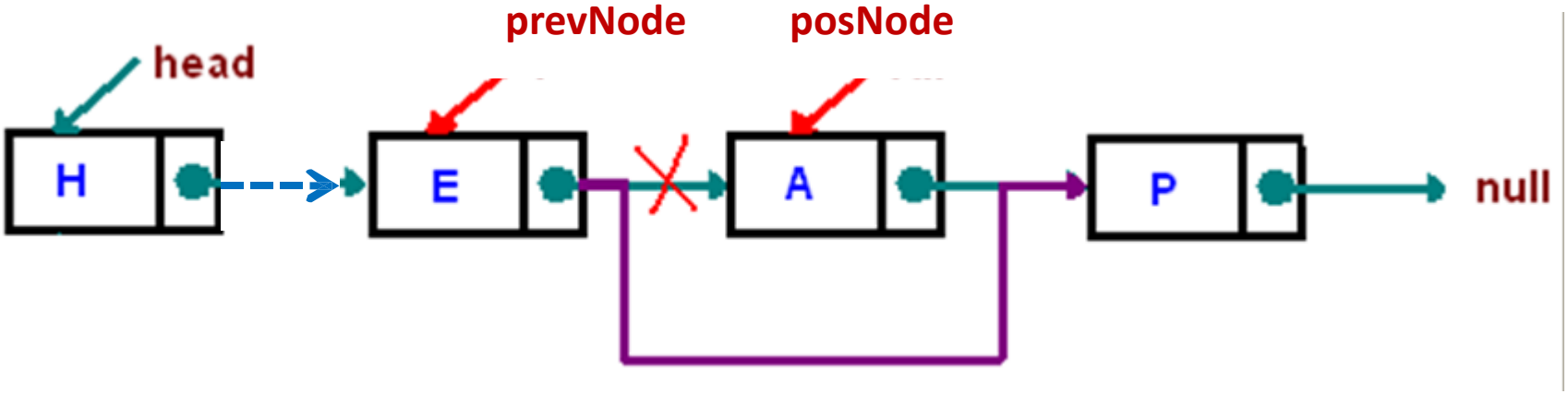
- remove(int pos): remove the node at the pos position from a linked list:
  - Travel to the pos position
  - Remove the pos node




```
prevNode=traversing(pos-1);
posNode=traversing(pos); //posNode=prevNode.getNext();
prevNode.setNext(posNode.getNext());
```

Singly Linked List[7]

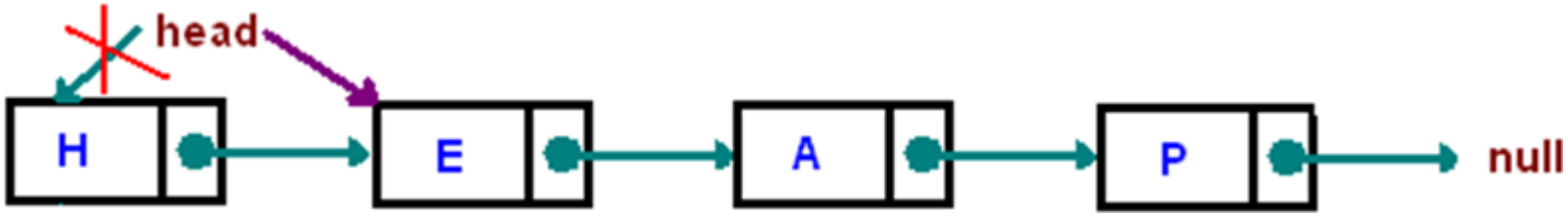
- remove(int pos): remove the node at the pos position from a linked list:
  - Travel to the pos position
  - Remove the pos node




```
prevNode=traversing(pos-1);
posNode=traversing(pos); //posNode=prevNode.getNext();
prevNode.setNext(posNode.getNext());
```

Singly Linked List[8]

- remove(int pos):
  - If pos equals 1
    - Remove the first node




```
head=head.getNext();
```

- removeAll():

```
head=null;
```

Singly Linked List[9]

- get(int pos):
  - Return the node at the pos position

```
return traversing(pos);
```

- getLength():

```
int count=0;
SLNode current=this.head;
while (current != null)
{
     count++;
     current=current.getNext();
}
return count;
```

List applications [1]

- Use list to solve problem related to order of a collection:
  - Data sequence management: mailing list, scrolling list (GUI)
  - Memory management: Hash table
  - And mathematical problem…
  - Linked lists are used as a building block for many other data structures, such as **stacks**, **queues** and their variations

List applications [2]

- Example: Representing Polynomials
  - Addition/subtraction /multiplication.. . of two polynomials.

$$P_1(x) = 2x^2+3x+7$$

$$P_2(x) = 3x^3+5x+2$$

$$P_1(x) + P_2(x) = 3x^3+2x^2+8x+9$$

- How can a list be used to represent that polynomial?

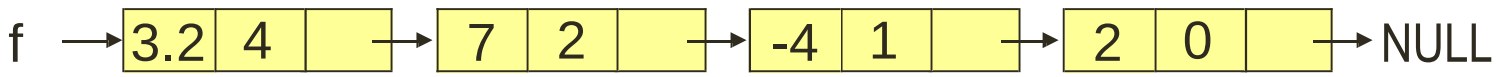
List applications [3]

For the polynomial:

$f(x) = 3.2x^4 + 7x^2 - 4x + 2$

We can represent it using a linked-list:

- Node's data contains a real number (coefficient) and an integer (degree).

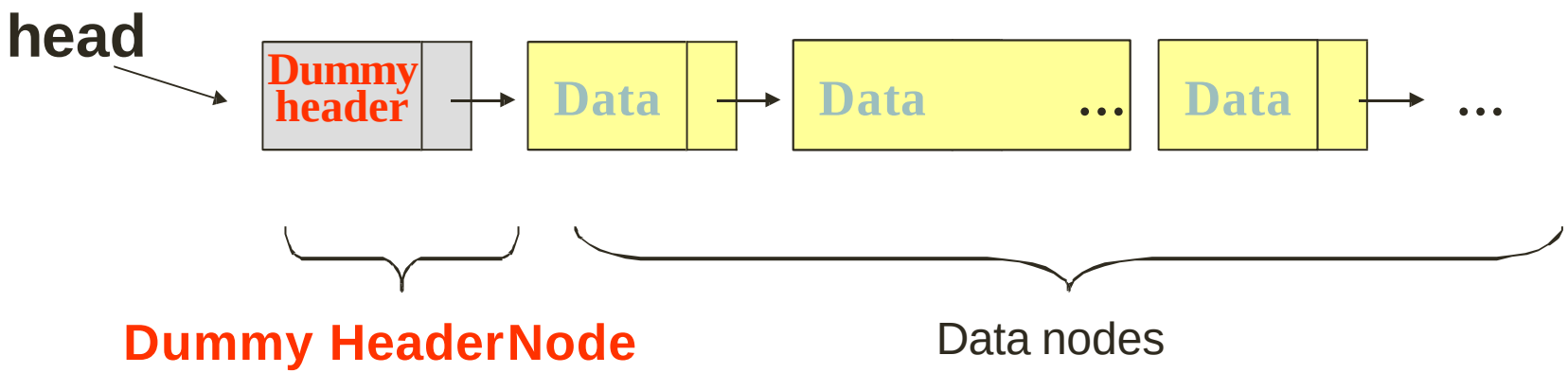


Other Linked-lists

- List with Dummy Header Node
- Circular Linked Lists
- Doubly Linked Lists

LL with Dummy Header Node

- Sometimes it is desirable to keep an extra node at the front of a list.

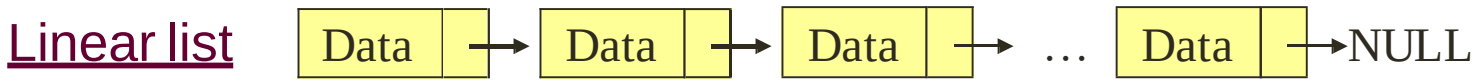


We don't care about the value of this node.

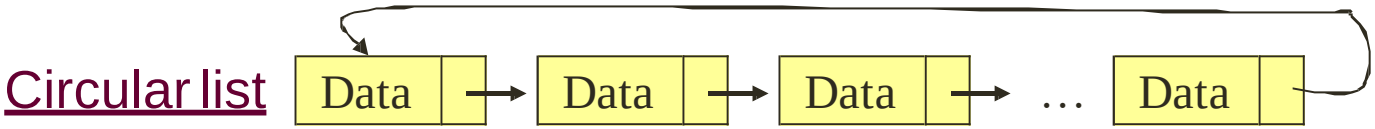
The value of these nodes are our data.

Circular Lists [1]

- In a linear list, the traversing order must be from head -> tail.



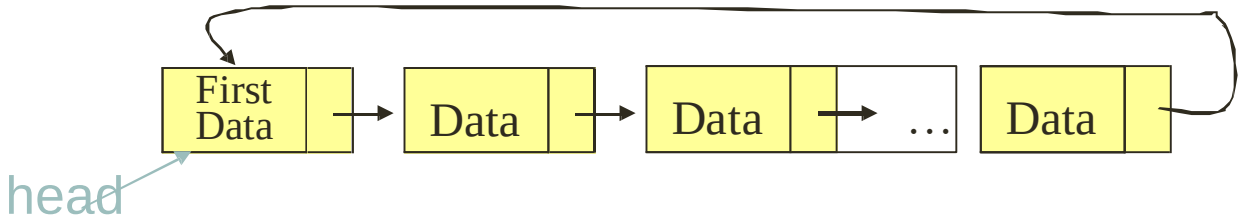
Consider the following implementation: **Circular list**



- From any point in such a list we can reach any other point in the list.

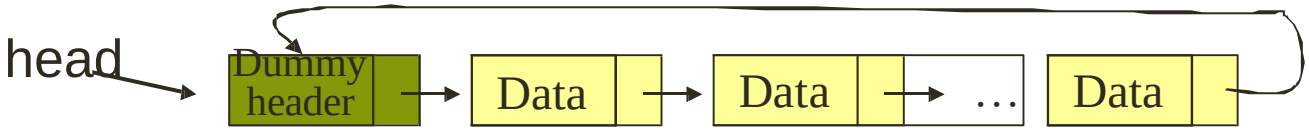
Circular Lists [2]

- To specify which node is the first one, we need a list head reference



- A null list head reference represents an empty circular list.

- We can also use a **dummy header node** that allows insertion and removal of an element conveniently from anywhere of the list.



An empty circular list:

Doubly Linked List [1]

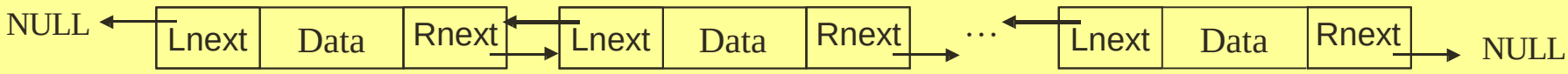
Some problems of singly linked lists:

- We can traverse the list in one direction only.
- To delete a node p, we need to know the predecessor of p.
- We can perform insertion after a given node in the list. But it is difficult to insert before a given node.

Doubly Linked List can solve these problems:
(Although it takes some extra memory space)

Linear, singly linked list

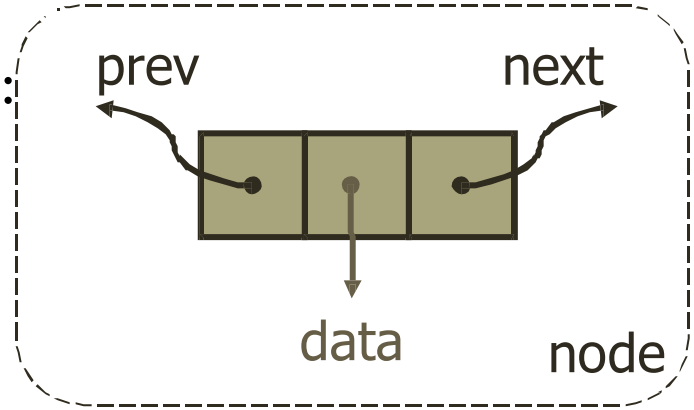
Linear, doubly linked list

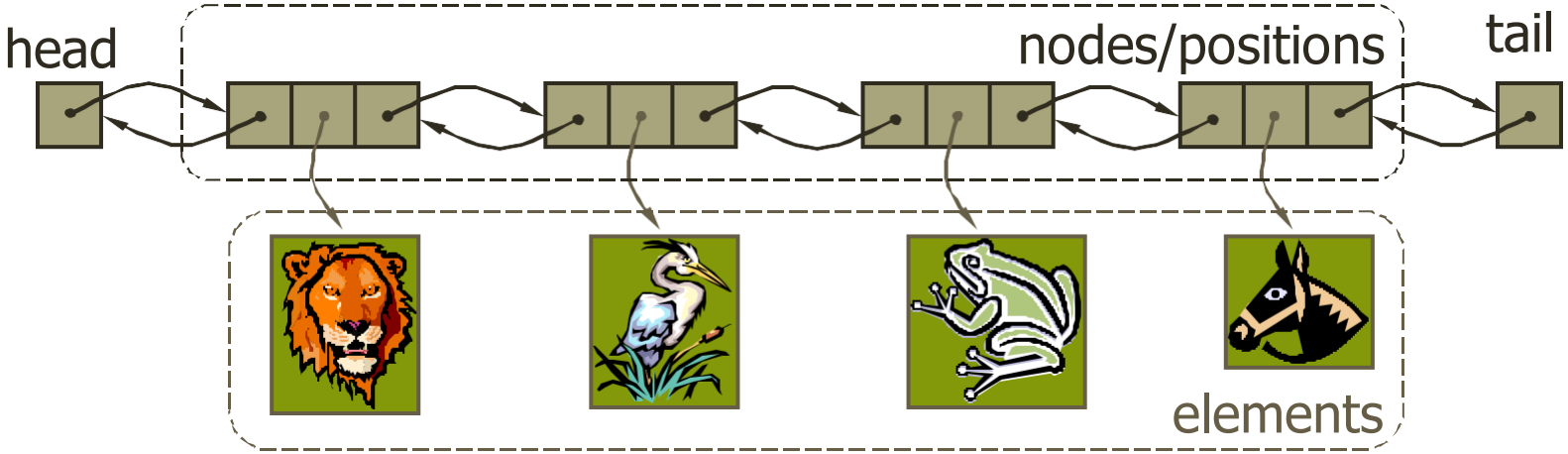


Doubly Linked List[2]

- A doubly linked list provides a natural implementation of the List ADT

- Nodes implement Position and store:
  - data
  - link to the previous node
  - link to the next node
- Special head and tail references

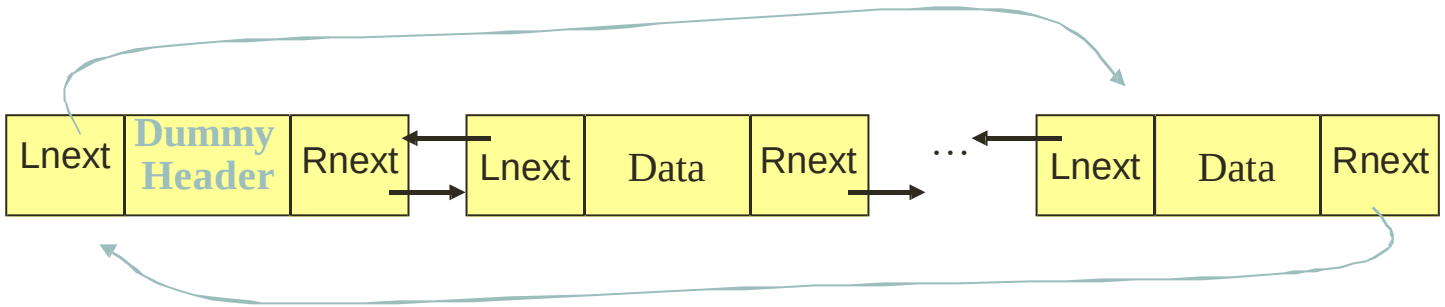




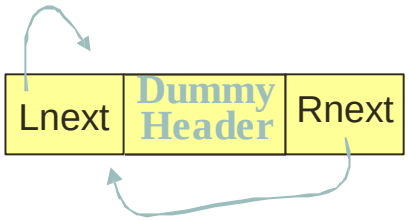
Doubly circular Linked List with Dummy Header Node

Doubly circular Linked List:

- A **dummy header node** can be included for easier manipulation:



- If the list is empty, both link fields of the header node point to itself.



Tutorial & next topic

- **Preparing for the tutorial:**
  - Practice with examples and exercises in Tutorial 5
- **Preparing for next topic:**
  - Read textbook chapter 3 (3.6 & 3.7) Stack and Queue.