Today's Material

- List ADT
  - Definition

- List ADT Implementation: LinkedList

List ADT

- What is a list?
  - An ordered sequence of elements A1, A2, …, AN

```
public class List {
  …
public:
  void add(int e);              // Add to the end (append)
  void add(int pos, int e);     // Add at a specific position
  void remove(int pos);         // Remove
  int  indexOf(int e);          // Forward Search
  int  lastIndexOf(int e);      // Backward Search
  bool clear();                 // Remove all elements
  bool isEmpty();               // Is the list empty?
  int  first();                 // First element
  int  last();                  // Last element
  int  get(int pos);            // Get at a specific position
  int  size();                  // # of elements in the list
};
```

Using List ADT

```
public static void main(String args[]){
 // Create an empty list object
  List list = new List();

  list.add(10);      // 10
  list.add(5);       // 10, 5
  list.add(1, 7);    // 10, 7, 5
  list.add(2, 9);    // 10, 7, 9, 5

  list.indexOf(7);  // Returns 1
  list.get(3);       // Return 5
  list.remove(1);    // 10, 9, 5
  list.size();       // Returns 3
  list.isEmpty();    // Returns false

  list.remove(0);    // 9, 5
  list.clear();      // empty list

}/* end-main */
```

Lists: Implementation

- Two types of implementation:
  - Array-Based - ArrayList
  - Linked - LinkedList

- We will compare worst case running time of ADT operations with different implementations

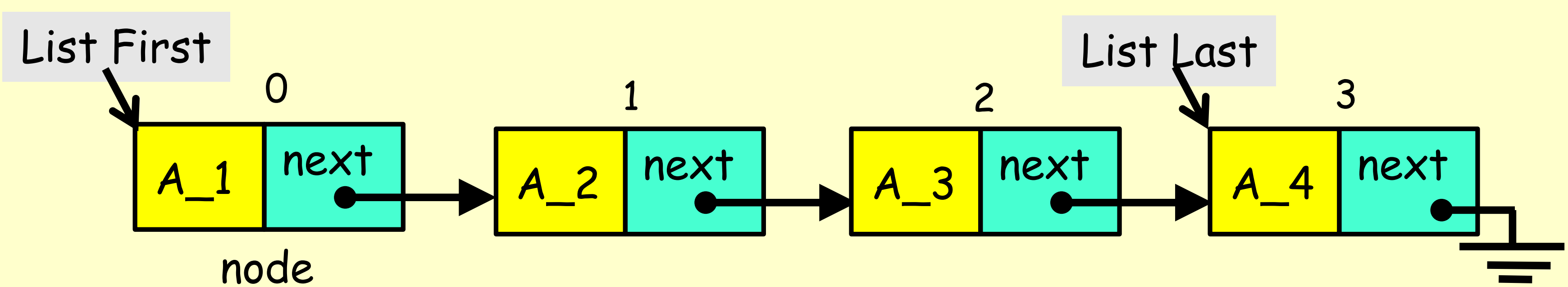
Lists: Array-Based Implementation

- Basic Idea:
  - Pre-allocate a big array of size MAX_SIZE
  - Keep track of first free slot using a variable N
  - Empty list has N = 0
  - Shift elements when you have to add or remove
  - What happens if the array is full?
    - Allocate a bigger array
    - Copy elements from the old array to the new one
    - Free up the space used by the old array
    - This requires a lot of memory copy operations!

0	1	2	3	………	N-1			MAX_SIZE
A_1	A_2	A_3	A_4	………	A_N-1			

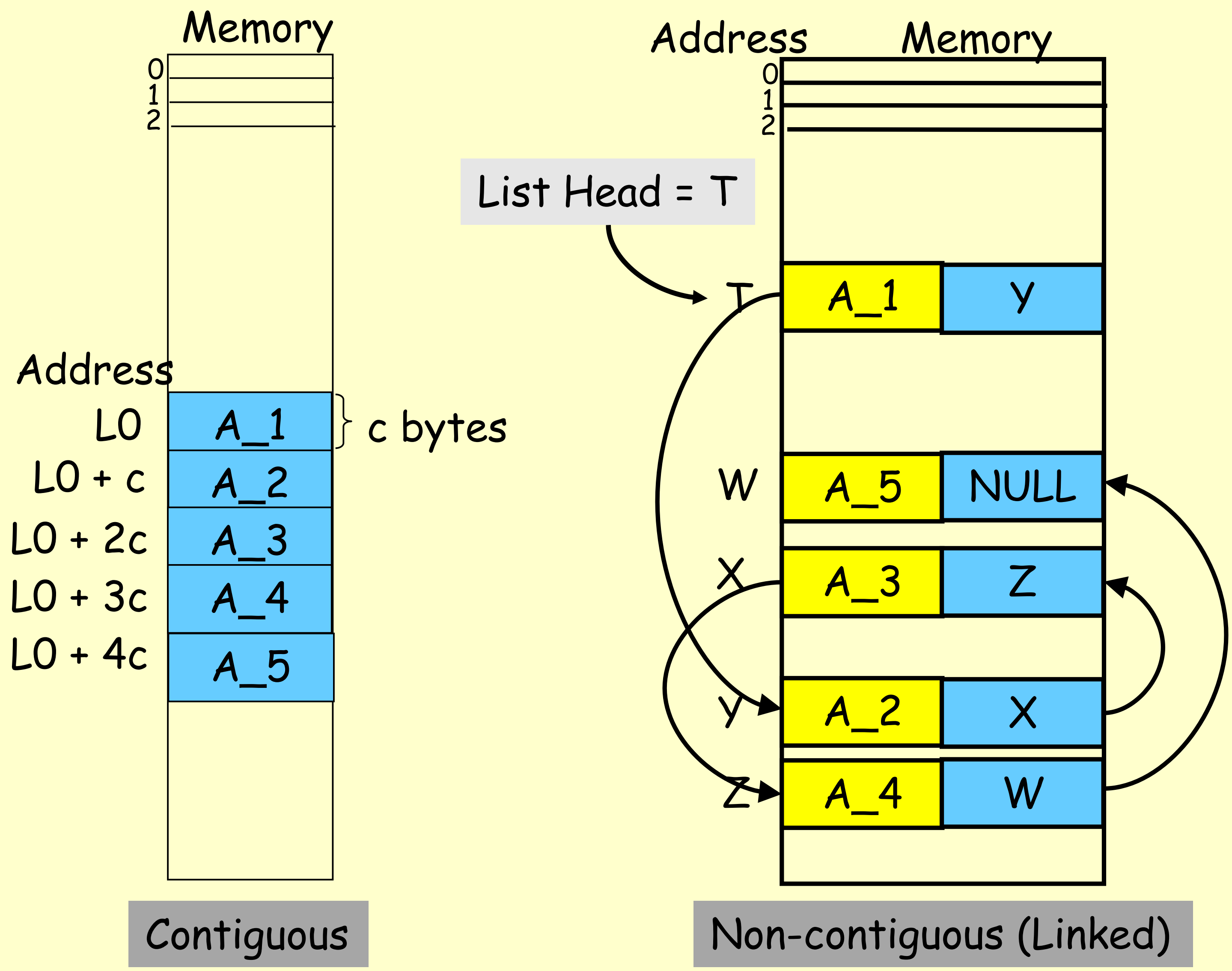
Lists: Linked Implementation

- Basic Idea:
  - Allocate one node per element
  - Nodes are NOT contiguous but are scattered in the memory
  - Each element keeps track of the location (address) of the next node that follows it
  - Need to know the location of the first node

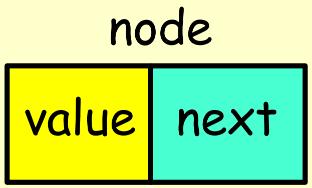


A list with 4 elements (nodes)

Lists: Contiguous vs Linked Implementation



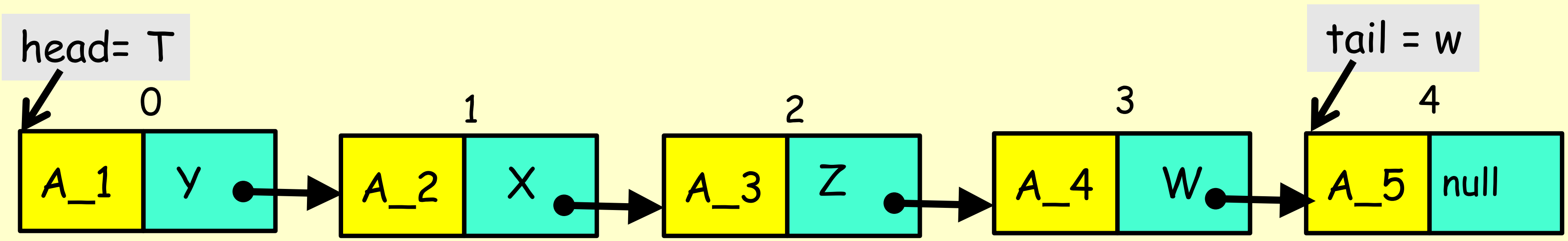
Linked Lists: Pictorial View



Pictorial view of a list node

```
class LinkedListNode {
  public ElementType value;
  public LinkedListNode next;
}
```

```
LinkedListNode head;
LinkedListNode tail;
```



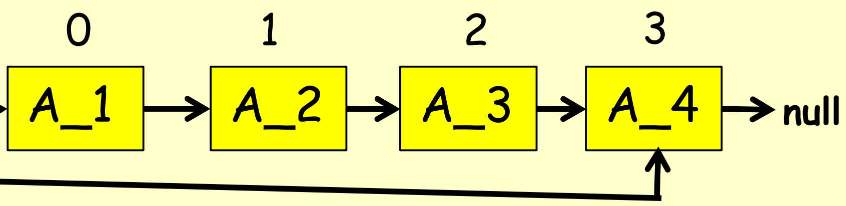
Pictorial view of the list shown on the previous page

Notice that T, W, X, Y and Z are arbitrary locations in memory

Linked List ADT- a Java implementation

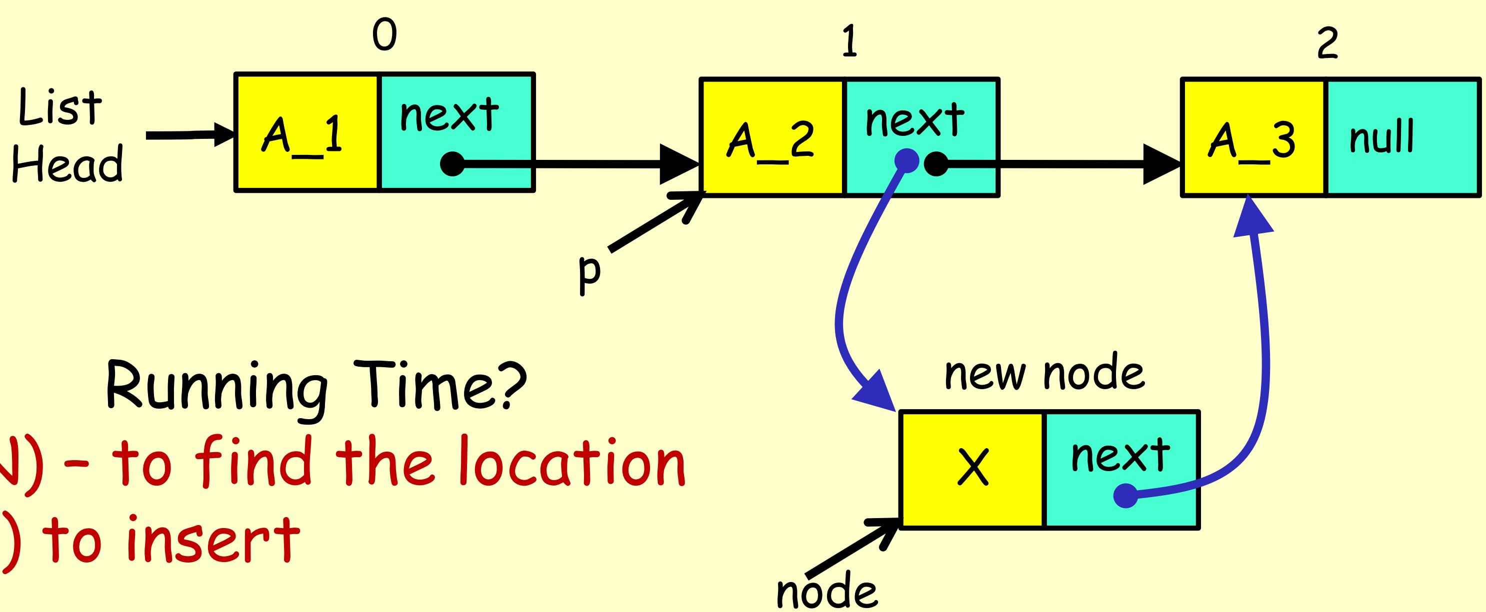
LinkedList ADT

```
class LinkedList {
private:
   LinkedListNode head;
   LinkedListNode tail;
   int noOfNodes;

public:
   LinkedList(){}

   void add(int pos, int e);
   void remove(int pos);
   int indexOf(int e);
   bool isEmpty();
   int first();
   int last();
   int get(int pos);
   int size();
};
```



Lists Operations: add

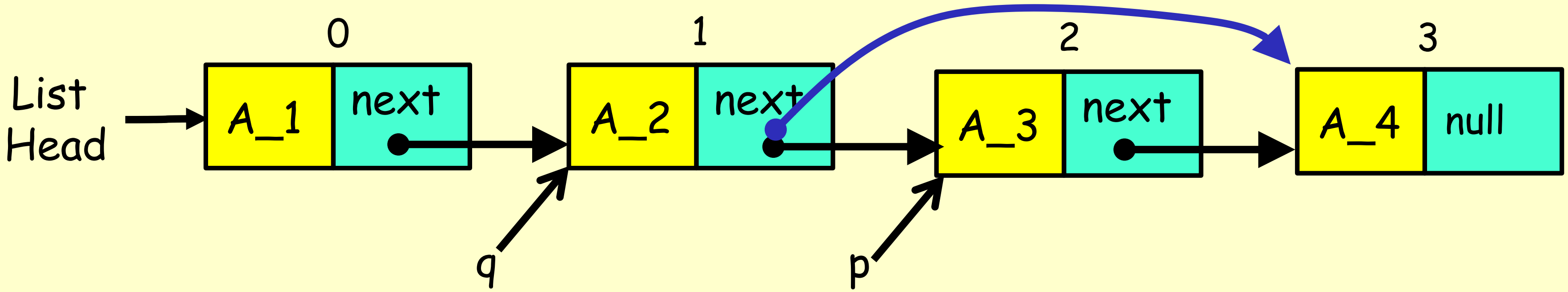
- add(Position P, ElementType E)
  - Example: add(2, X): Insert X at position 2
  - Algorithm:
    (1) Find where X needs to be inserted (after p)
    (2) Create a new node containing X
    (3) Update next pointers



Running Time?
O(N) – to find the location
O(1) to insert

Lists Operations: remove

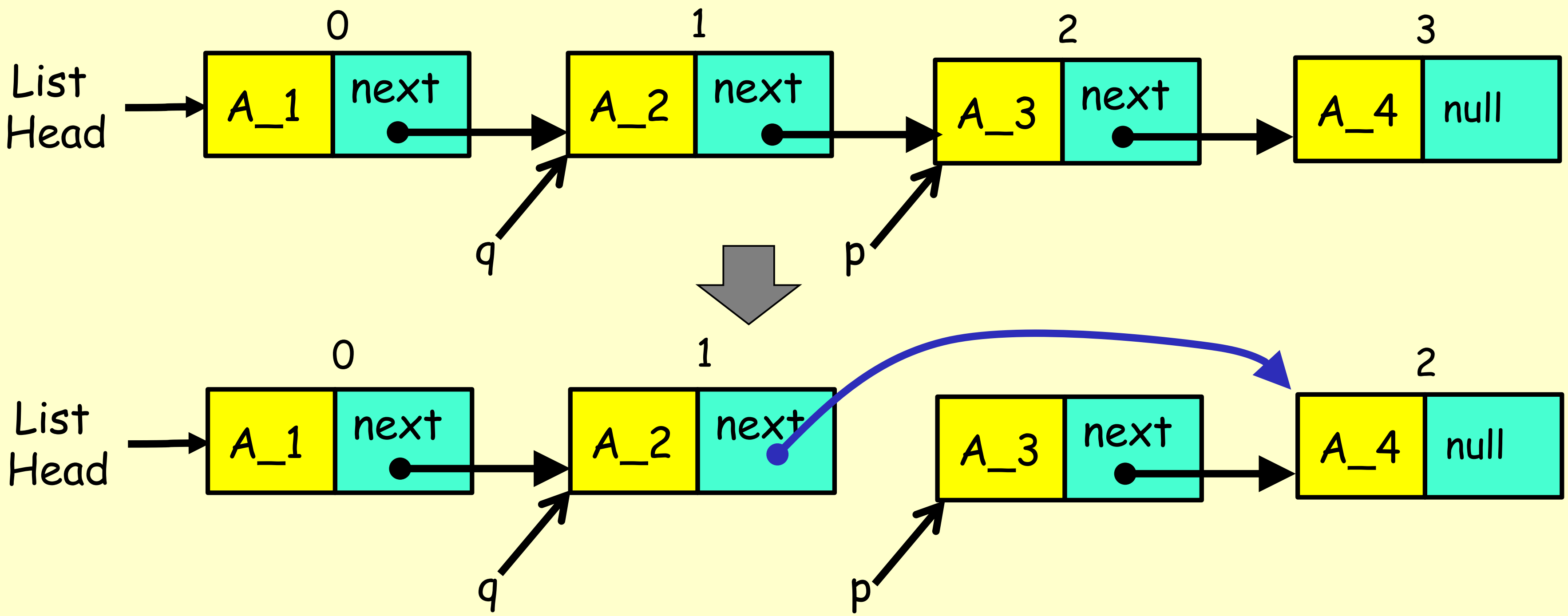
- remove(Position P)
  - Example: remove(2): Delete element at position 2
  - First we need to find the node to delete



  - Can we delete the node pointed to by p?
  - Need a pointer to the previous node
    - While finding the node to delete, keep track of the previous node (q trails p as we go over the list)
  - Now adjust the next pointers – How?
    - q.next = p.next

Lists Operations: remove

- remove(Position P)
  - Example: remove(2): Delete element at position 2

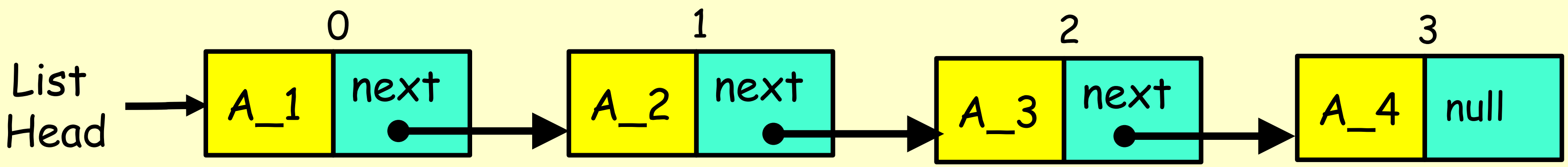


Running Time?

O(N) – to find the node, O(1) to delete

Lists Operations: indexOf

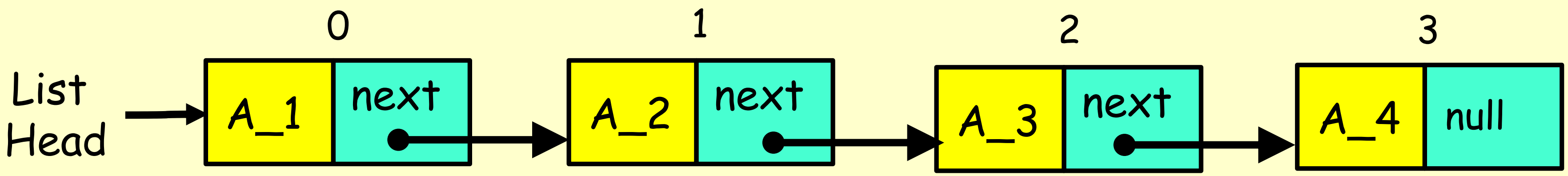
- indexOf(ElementType E)
  - Example: indexOf(X): Search X in the list



- Must do a linear search
  - Running time: O(N)

Lists Operations: isEmpty

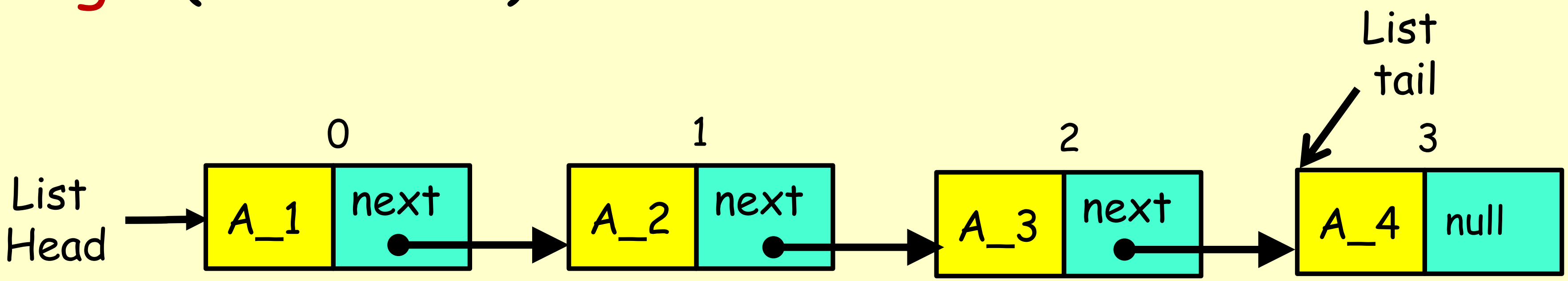
- isEmpty()
  - Returns true if the list is empty



- Trivial – Return true if head == NULL
  - Running time: O(1)

Lists Operations: first, last, get

- first()
- last()
- get(Position K)



- first – Running time: O(1)
- last – Running time: O(1) – If we keep a tail ptr
- get – Running time: O(N)

Caveats with Linked Lists

- Whenever you break a list, your code should fix the list up as soon as possible
  - Draw pictures of the list to visualize what needs to be done

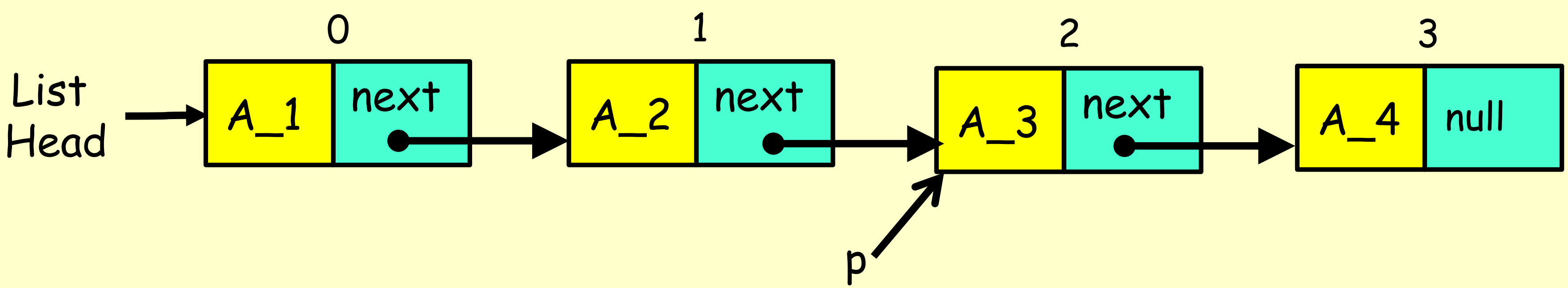
- Pay special attention to boundary conditions:
  - Empty list
  - Single node– same node is both first and last
  - Two nodes– first, last, but no middle nodes
  - Three or more nodes– first, last, and middle nodes

Lists: Running time comparison

Operation	Array-based List	Linked List
add	O(N)	O(N)
add(to the end)	O(N)	O(1)
remove	O(N)	O(N)
indexOf	O(N)	O(N)
isEmpty	O(1)	O(1)
first	O(1)	O(1)
last	O(1)	O(1)
get	O(1)	O(N)
size	O(1)	O(1)

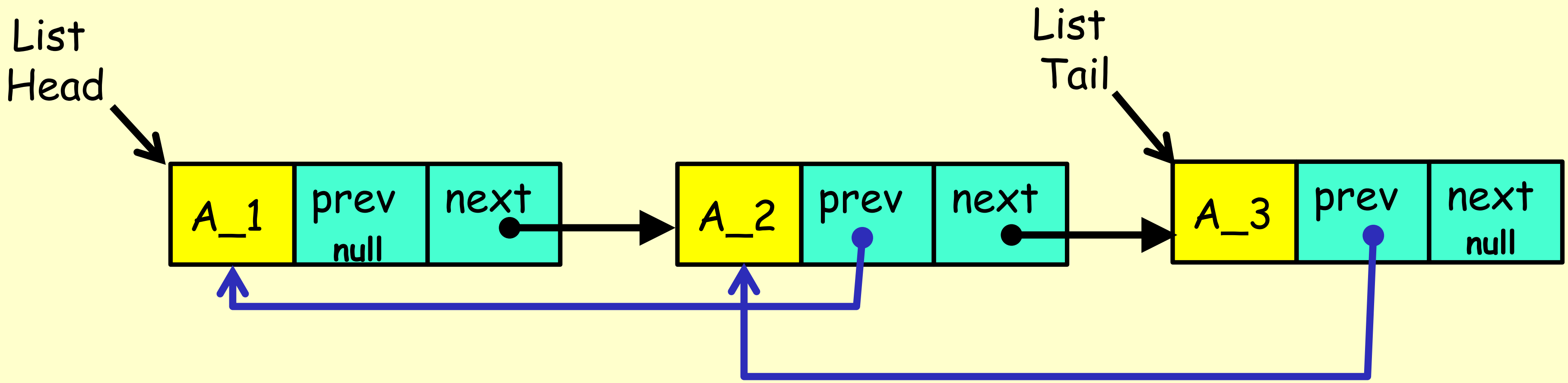
More on remove

- What if you already have a pointer to the node to delete?
  - remove(LinkedListNode *p);



- Still need a pointer to the previous node
  - Makes the running time O(N)
  - Can we make this operation faster?
  - Yep: Also keep a pointer to the previous node!
    - Called a doubly linked list

Doubly Linked Lists

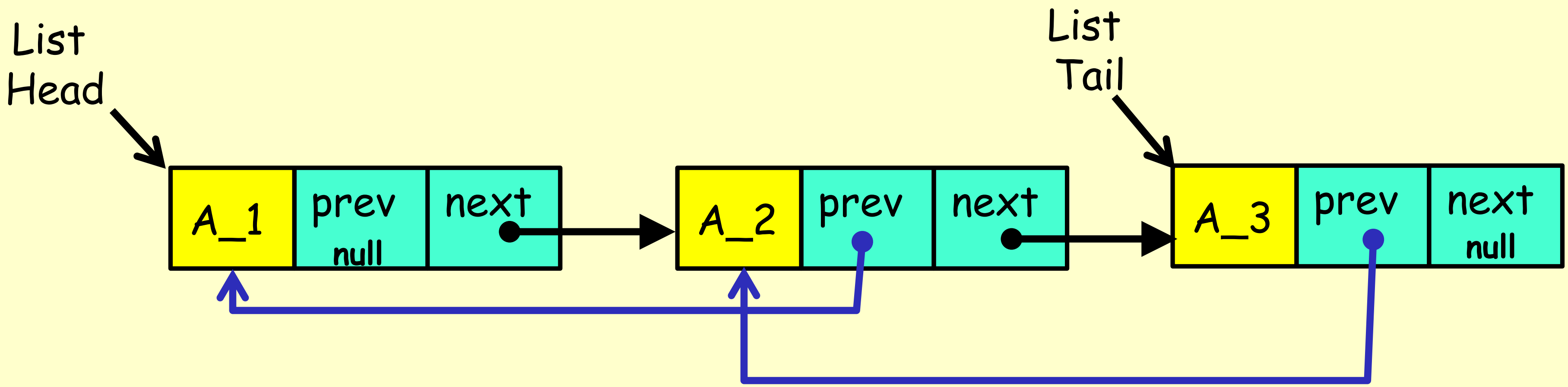


Java Declarations

```
class DoublyLinkedListNode {
  public ElementType value;
  public DoublyLinkedListNode next;
  public DoublyLinkedListNode prev;
}
```

```
DoublyLinkedListNode head;
DoublyLinkedListNode tail;
```

Doubly Linked Lists



- Advantages:
  - remove(LinkedListNode p) becomes O(1)
  - previous(LinkedListNode p) becomes O(1)
  - Allows going over the list forward & backwards
- Disadvantages:
  - More space (double the number of pointers at each node)
  - More book-keeping for updating two pointers at each node

Application of Lists: Polynomials

- Problem with Array Lists: Sparse Polynomials
  - E.g. 10X^3000+ 4X^2+ 7
  - Waste of space and time (C_i are mostly 0s)

- Solution?
  - Use singly linked list, sorted in decreasing order of exponents