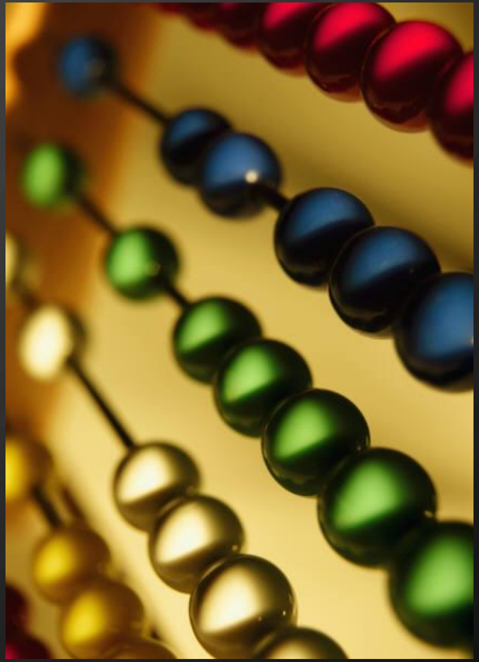
WELCOME TO CSCE 221—DATA STRUCTURES

SYLLABUS

CH3. FUNDAMENTAL DATA STRUCTURES

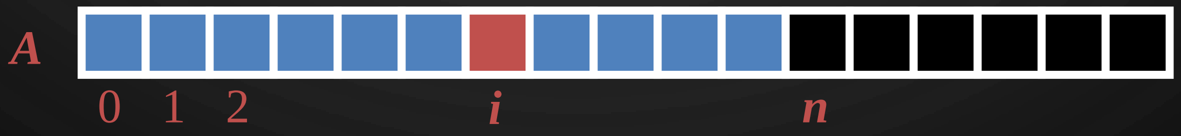
ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN JAVA, GOODRICH, TAMASSIA AND GOLDWASSER (WILEY 2016)

ARRAYS



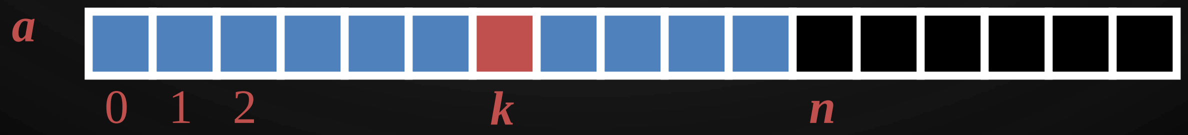
ARRAY DEFINITION

- An **array** is a sequenced collection of variables all of the same type. Each variable, or **cell**, in an array has an **index**, which uniquely refers to the value stored in that cell. The cells of an array, A, are numbered $0, 1, 2$, and so on.
- Each value stored in an array is often called an **element** of that array.



ARRAY LENGTH AND CAPACITY

- Since the length of an array determines the maximum number of things that can be stored in the array, we will sometimes refer to the length of an array as its **capacity**.
- In Java, the length of an array named `a` can be accessed using the syntax `a.length`. Thus, the cells of an array, `a`, are numbered 0, 1, 2, and so on, up through `a.length−1`, and the cell with index k can be accessed with syntax `a[k]`.



DECLARING ARRAYS (FIRST WAY)

- The first way to create an array is to use an assignment to a literal form when initially declaring the array, using a syntax as:

$$elementType[\,]\ arrayName = \{initialValue_0, initialValue_1, \ldots, initialValue_{N-1}\};$$

- The `elementType` can be any Java base type or class name, and `arrayName` can be any valid Java identifier. The initial values must be of the same type as the array.

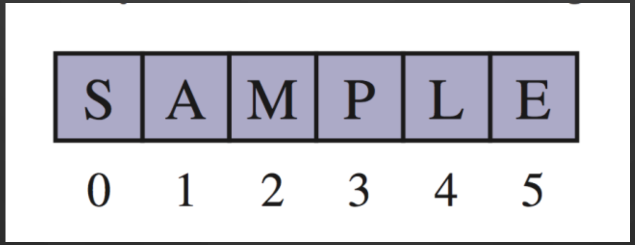
DECLARING ARRAYS (SECOND WAY)

- The second way to create an array is to use the `new` operator.
  - However, because an array is not an instance of a class, we do not use a typical constructor. Instead we use the syntax:

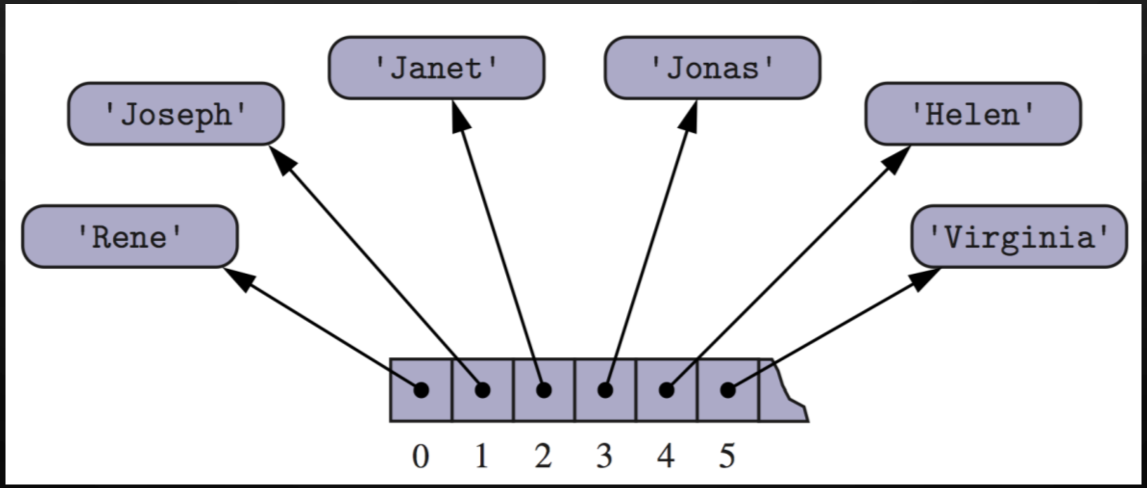
    `new elementType[length]`
- `length` is a positive integer denoting the length of the new array.
- The `new` operator returns a reference to the new array, and typically this would be assigned to an array variable.

ARRAYS OF CHARACTERS OR OBJECT REFERENCES

- An array can store primitive elements, such as characters.

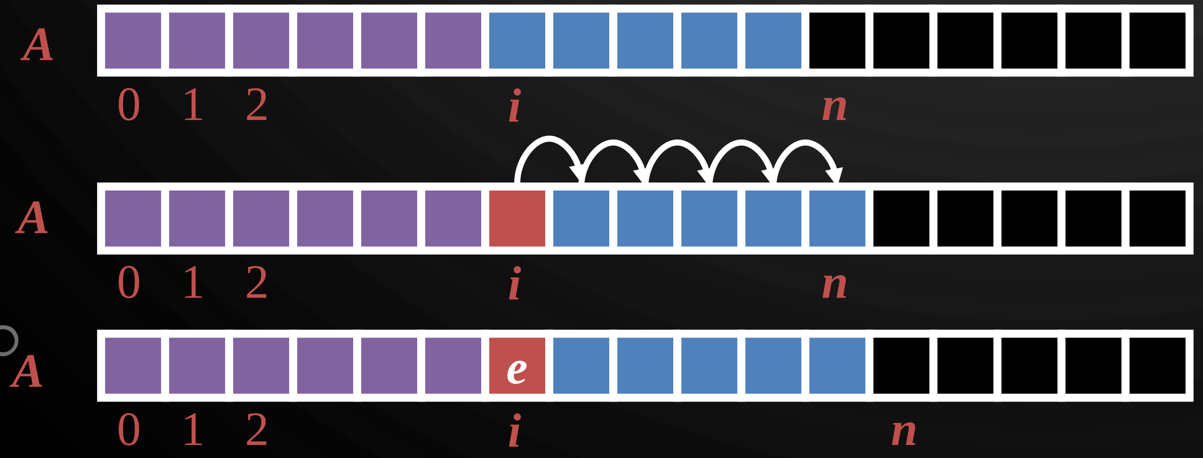


- An array can also store references to objects.



ADDING AN ENTRY

- To add an entry e into array A at index i, we need to make room for it by shifting forward the $n-i$ entries $A[i], \dots, A[n-1]$



Add

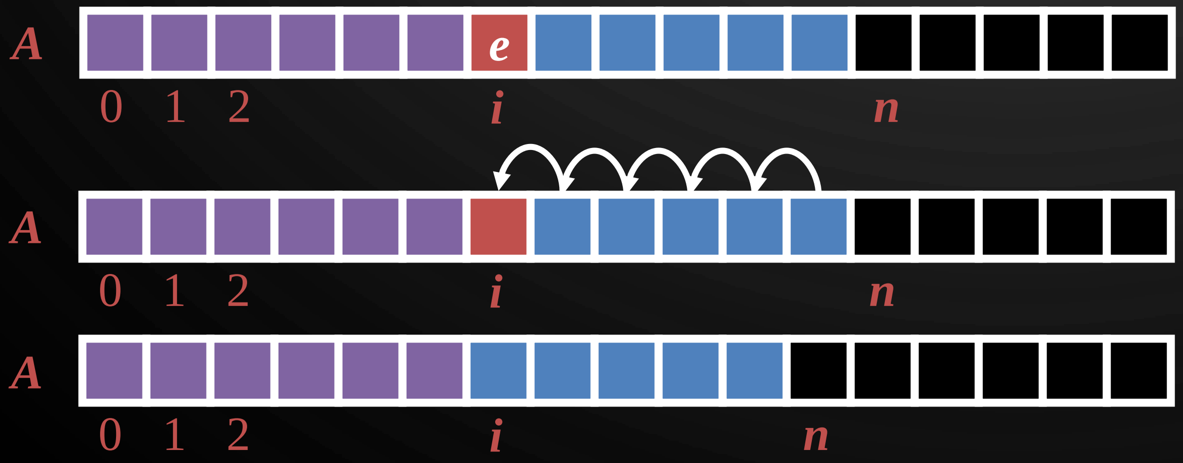
```
Input: Array $A$,
   index $i$, element $e$
1. for $k \leftarrow n, n-1, \dots, i+1$
2.     $A[k-1] \leftarrow A[k]$
3. $A[i] \leftarrow e$
4. $n \leftarrow n+1$
```

REMOVING AN ENTRY

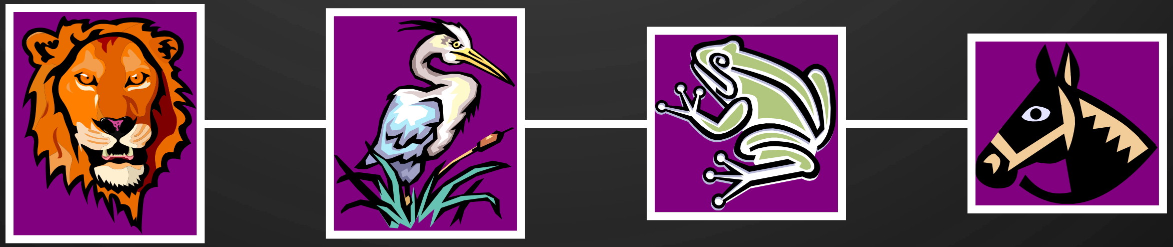
- To remove the entry e at index i, we need to fill the hole left by e by shifting backward the $n - i - 1$ elements $A[i + 1], \ldots, A[n - 1]$

Add

```
Input: Array $A$,
   index $i$, element $e$
1. for $k \leftarrow i + 1, \ldots, n - 1$
2.     $A[k - 1] \leftarrow A[k]$
3. $A[n - 1] \leftarrow null$
4. $n \leftarrow n - 1$
```

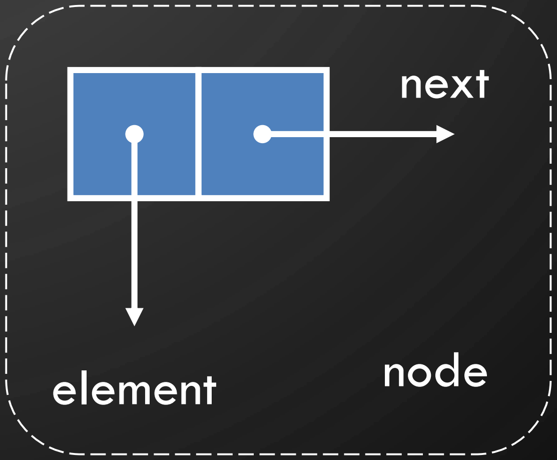


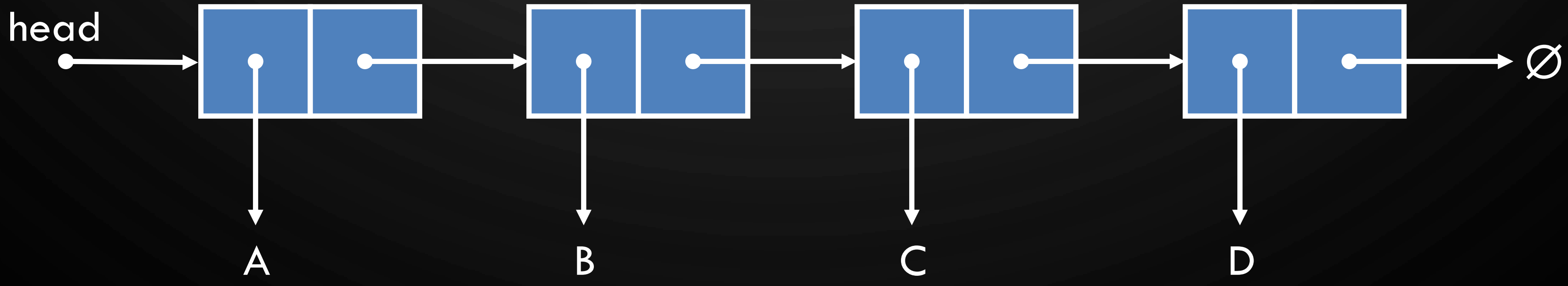
SINGLY LINKED LISTS



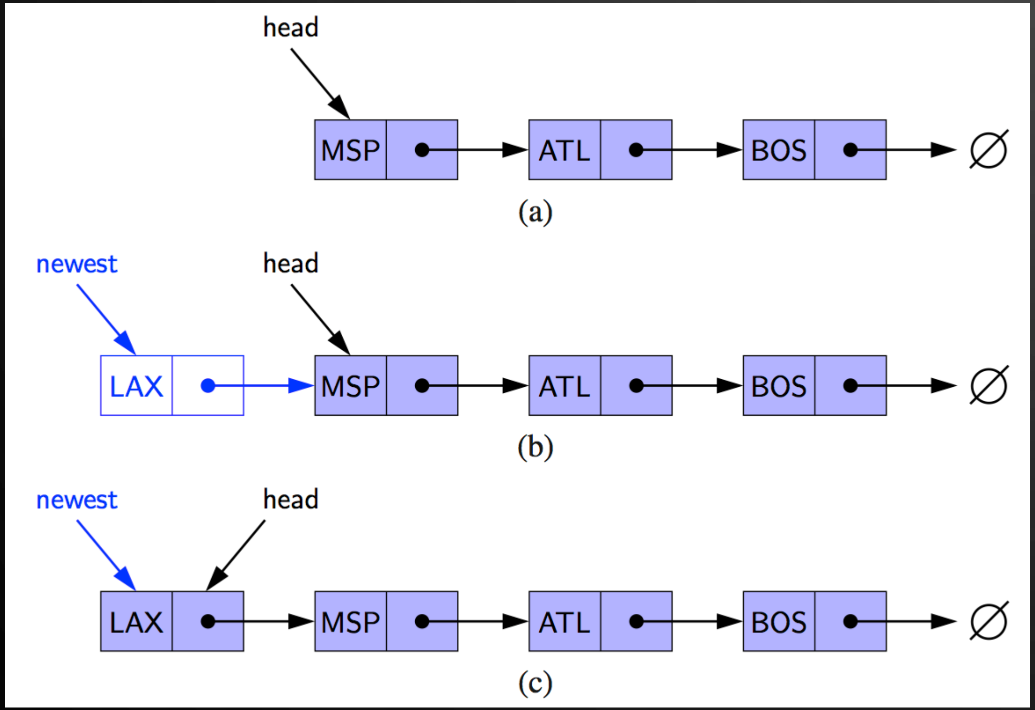
SINGLY LINKED LIST

- A **singly linked list** is a concrete data structure consisting of a sequence of nodes, starting from a head pointer
- Each node stores
    - element
    - link to the next node





INSERTING AT THE HEAD



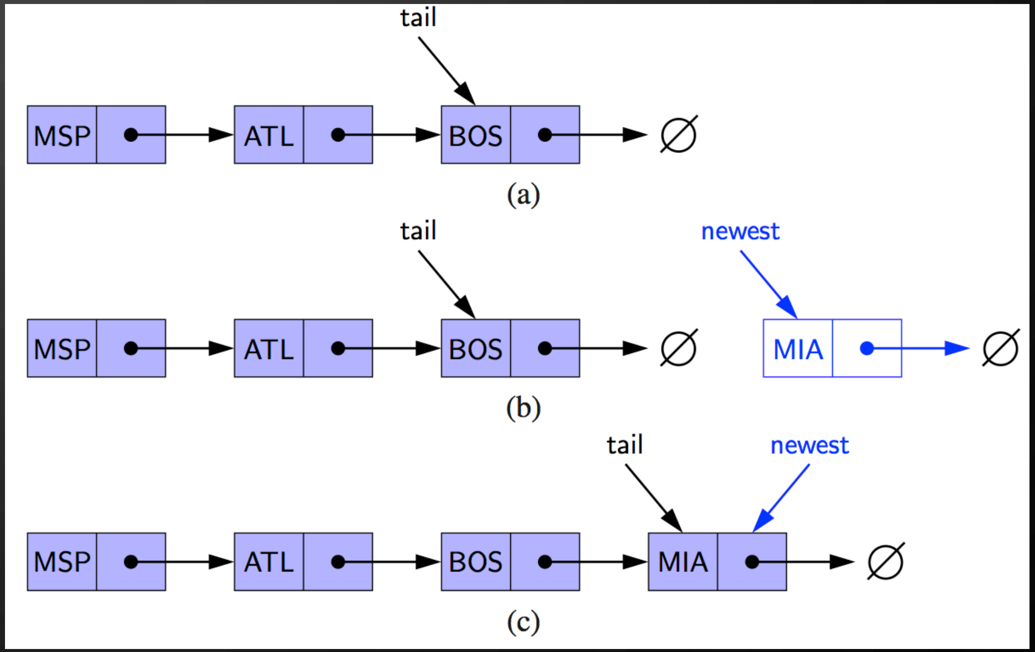
AddFirst

```
Input: List l, Element e
1. $n \leftarrow Node(e)$ //Allocate new node $n$ to contain element $e$
2. $n.next \leftarrow l.head$ //Have new node point to old head
3. $l.head \leftarrow n$ //Update head to point to new node
```

INSERTING AT THE TAIL

AddLast

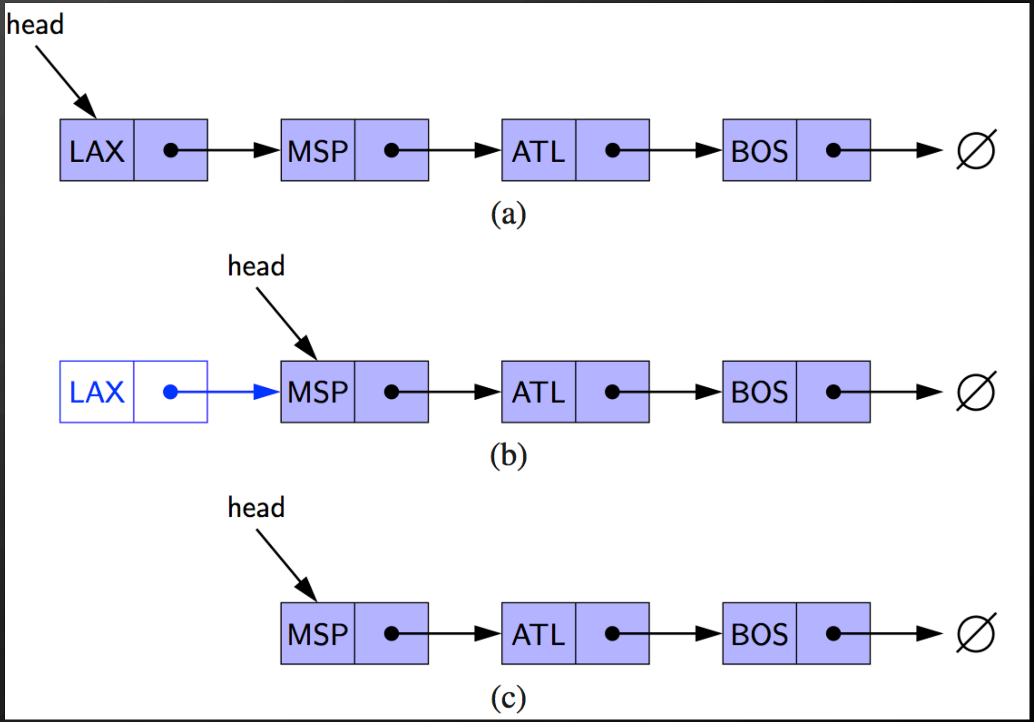
```
Input: List $l$, Element $e$
1. $n \leftarrow Node(e)$ //Allocate a new node to contain element $e$
2. $n.next \leftarrow null$ //Have new node point to null
3. $l.tail.next \leftarrow n$ //Have old last node point to new node
4. $tail \leftarrow n$ //Update tail to point to new node
```



REMOVING AT THE HEAD

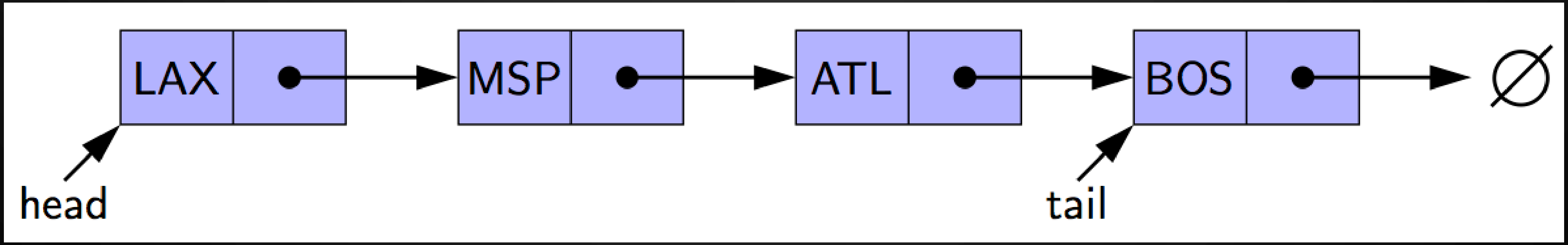
RemoveFirst

```
Input: List $l$
1. $l.head \leftarrow l.head.next$ //Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node
```

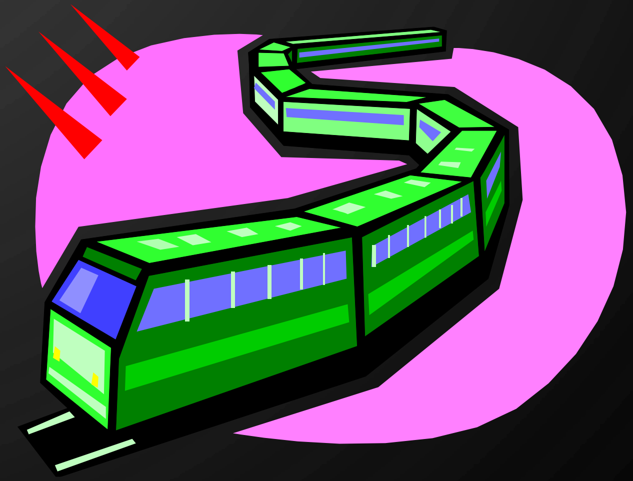


REMOVING AT THE TAIL

- 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

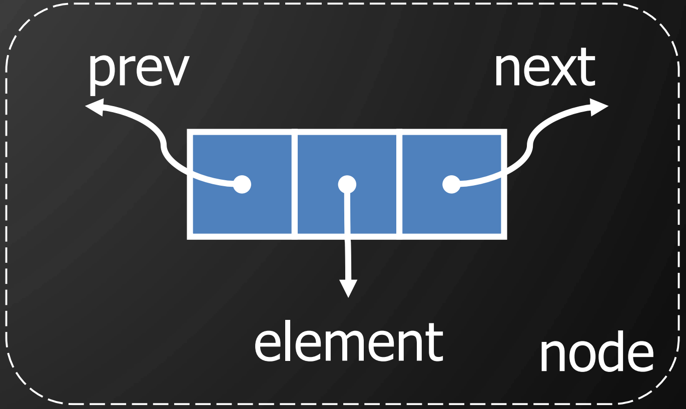


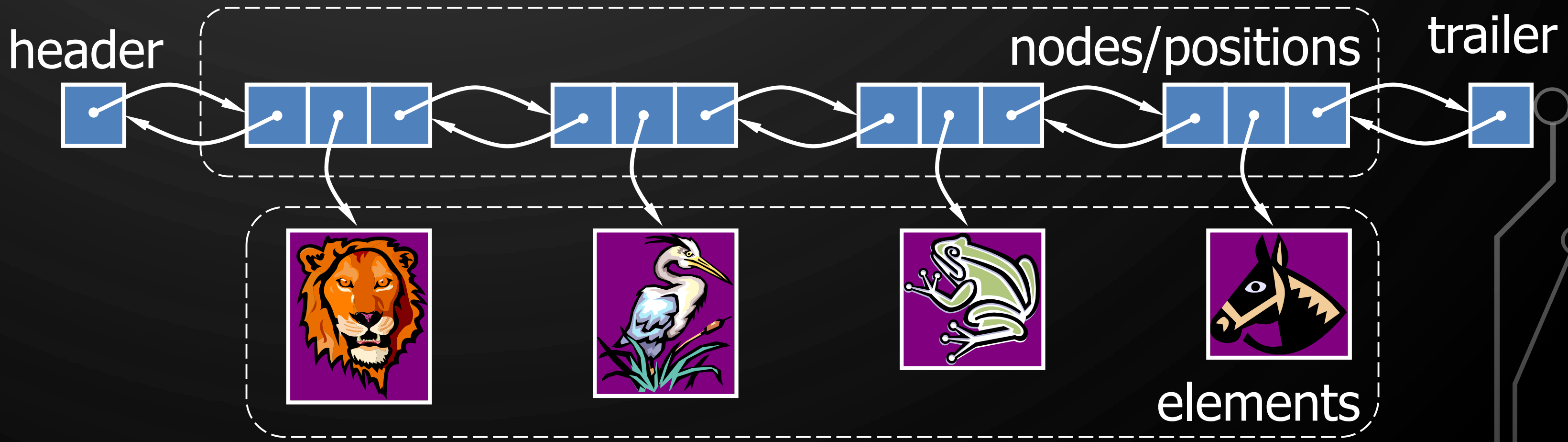
DOUBLY LINKED LISTS



DOUBLY LINKED LIST

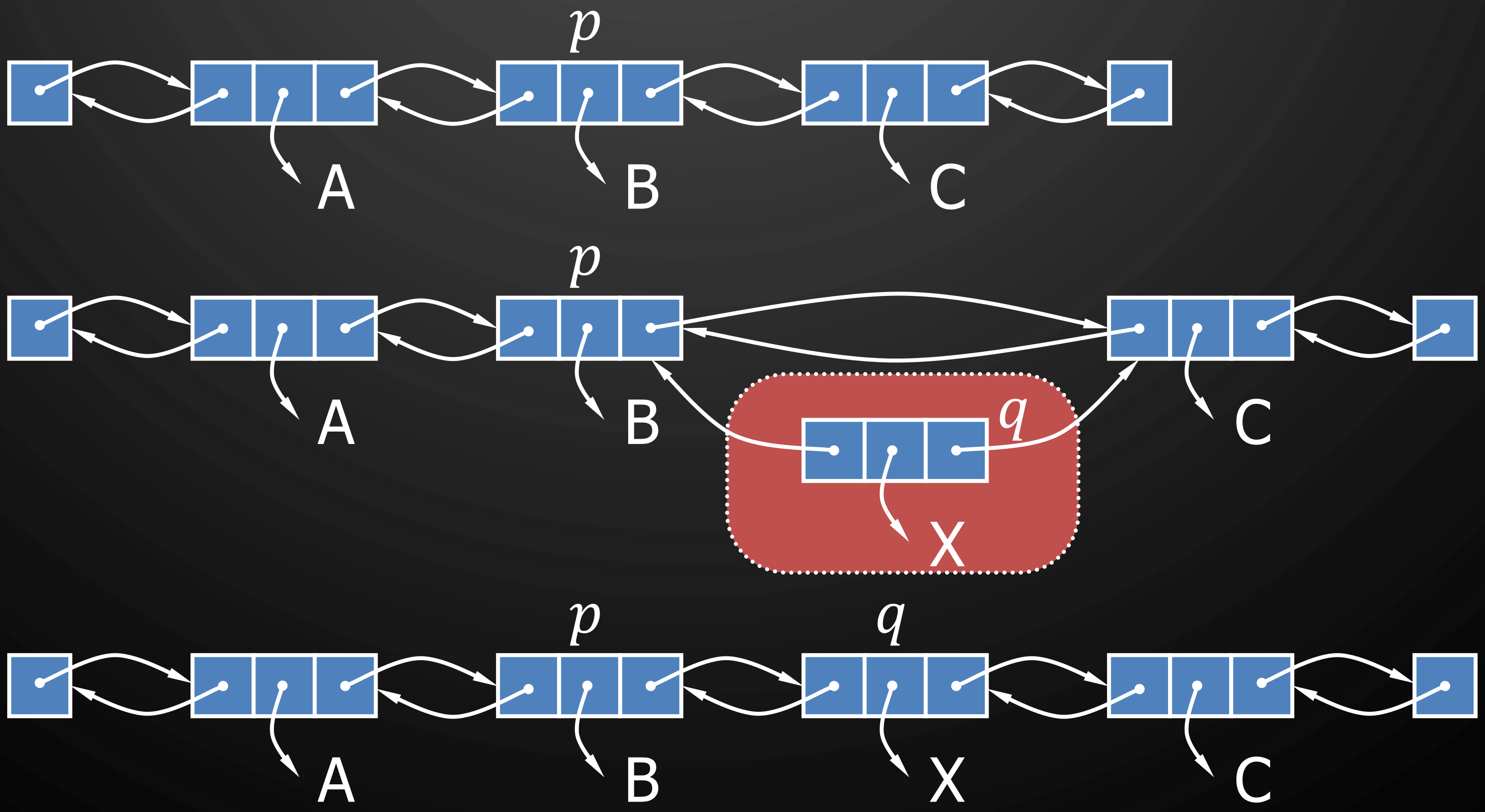
- A **doubly linked list** can be traversed forward and backward
- Nodes store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





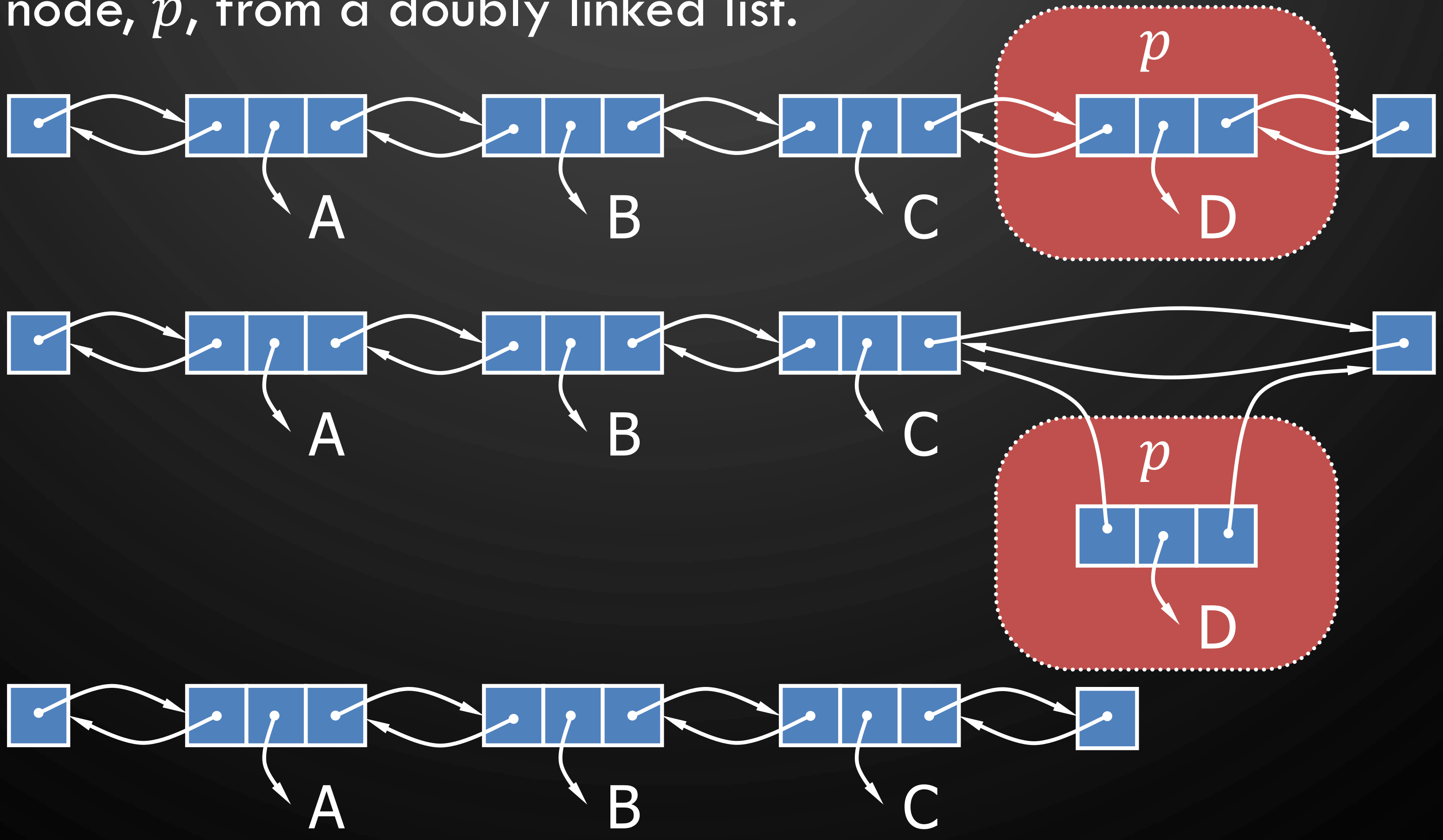
INSERTION

- Insert a new node, q, between p and its successor.



DELETION

- Remove a node, p, from a doubly linked list.



p
A
B
C
D
A
B
C
p
D
A
B
C