FALL 2018 CISC 311

# OBJECT & STRUCTURE & ALGORITHM I

– Chapter 4: Array Lists and Linked Lists

Department of Math & CS

# Outline

- Abstract data type (ADT)
- The array list ADT
- The linked list ADT
  - Singly linked list
  - Variants
    - Doubly linked list
    - Circularly linked list
- Implementations

# Abstract Data Types

- **Abstraction** is to distill a system to its most fundamental parts
- Applying the abstraction paradigm to the design of data structures gives rise to **abstract data types** (ADTs)
- An ADT is a model of a data structure that specifies the **type** of data stored, the **operations** supported on them, and the types of **parameters** of the operations
- An ADT specifies what each operation does, but not how it does it
- The collective set of behaviors supported by an ADT is its **public interface**

# Abstract Data Types (cont'd.)

- All the high-level programming languages already have concrete implementation of arrays
- Common data structures:

Data structures
- Array
- ADT
  - Array List (vector in C++)
  - Linked list
  - Stack
  - Queue
  - Deque
  - Priority queue
  - Map
  - Hash Table
  - Tree
  - Graph

# ADT vs. Data Structures

- Abstract data types are mathematical models of data types, where the data type is defined by how it is used, i.e., from the point of view of the user
- These define the operations to be performed on data, and what the expected behavior of those operations are

- Data structures are concrete representations of data from the point of view of an implementor
- This specifies the actual implementation of the structure in code to meet the expected behavior

# How We Study Data Structures in this Course

- ADT: logical model which defines data and operations
  - Abstract view
  - Operations
  - Complexity of each operation
- Implementation

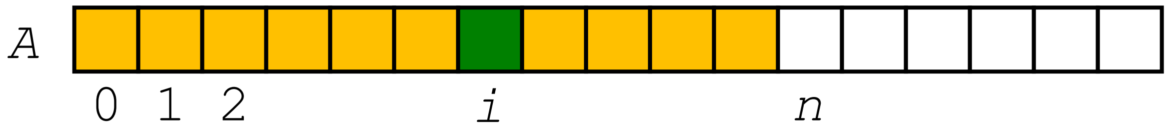
# The Array List ADT

- The array list ADT:
  - Empty list has size 0
  - Store a given number of elements of any type
  - Specify elements’ data type
  - Read elements by position
  - Write/modify element at a position
  - Count the number of elements in the list
  - Insert/remove an element into/from the list at any position
  - Grow/shrink as needed to store data during program runtime
- Implementation:  array list in Java / vector in C++
  - Array-based implementation

| Method | Description |
| --- | --- |
| `size()` | Returns the number of elements in the array list |
| `isEmpty()` | Returns a boolean indicating whether the array list is empty |
| `get(i)` | Returns the element of the list having index *i*; and error condition occurs if *i* is not in range [0, `size`()-1] |
| `set(i, e)` | Replaces the element at index *i* with *e*, and returns the old element that was replaced; an error condition occurs if *i* is not in range [0, `size`()-1] |
| `add(i, e)` | Inserts a new element *e* into the list so that it has index *i*, moving all subsequent elements one index later in the list; an error condition occurs if *i* is not in range [0, `size`()] |
| `remove(i)` | Removes and returns the element at index *i*; moving all subsequent elements one index earlier in the list; an error condition occurs if *i* is not in range [0, `size`()-1] |

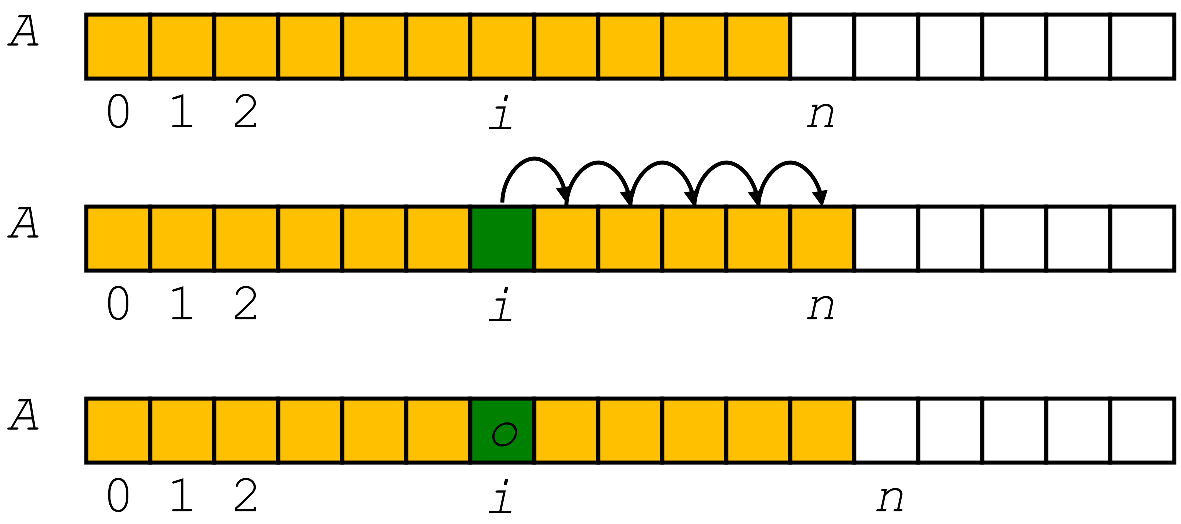
# Array Lists

- An obvious choice for implementing the list ADT is to use an array, $A$, where $A[i]$ stores (a reference to) the element with index $i$
- With a representation based on an array $A$, the `get(i)` and `set(i,e)` methods are easy to implement by accessing $A[i]$ (assuming $i$ is a legitimate index)



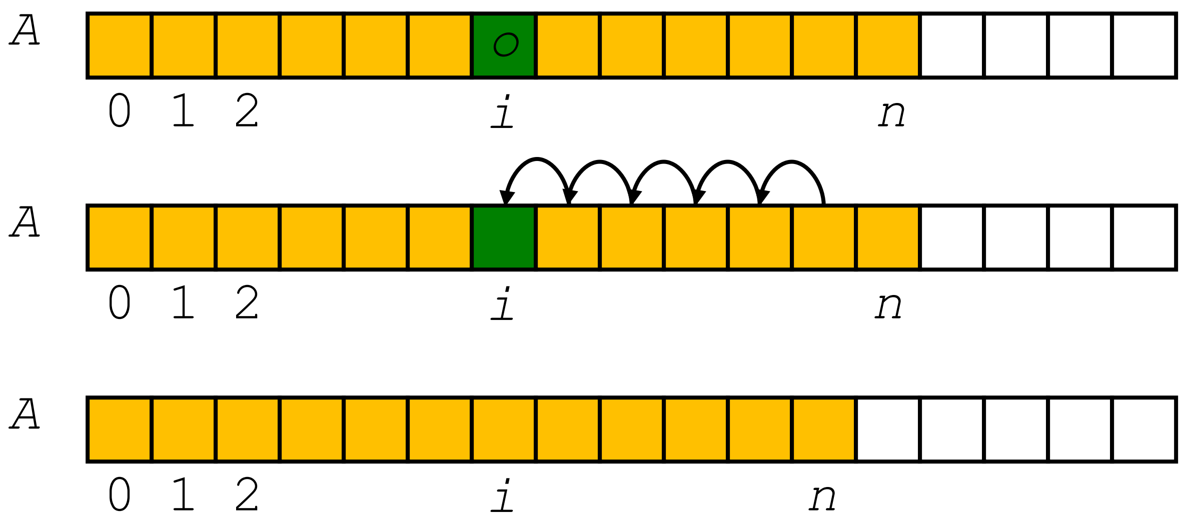
# Insertion

- In an operation `add(`$i$`,` $o$`)`, we need to make room for the new element by shifting forward the $n-i$ elements $A[i]$, …, $A[n-1]$
- In the worst case ($i$ = 0), this takes $O(n)$ time



# Element Removal

- In an operation `remove(i)`, we need to fill the hole left by the removed element by shifting backward the $n-i-1$ elements $A[i+1]$, …, $A[n-1]$
- In the worst case ($i$ = 0), this takes $O(n)$ time

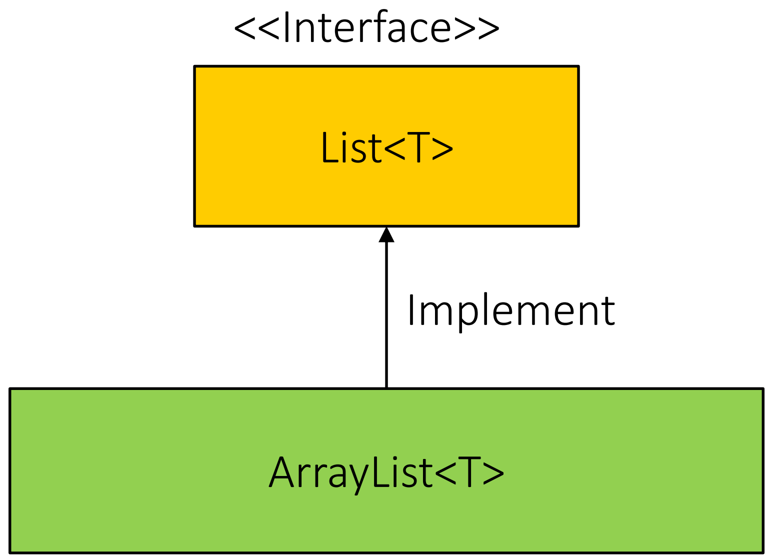


# Performance

- In an array-based implementation of a dynamic list:
  - The space used by the data structure is $O(n)$
  - Indexing the element at `i` takes $O(1)$ time
  - `Add` and `remove` run in $O(n)$ time
- In an add operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one.

| | Access/Update | Search | Insertion | Deletion |
|---|---|---|---|---|
| Array list | $O(1)$ | $O(n)$ | $O(n)$ | $O(n)$ |

# Implementation



# A Simple Array-Based Implementation

- Implement an array list with an array of fixed size (static implementation):
  - Create an array with max length
  - Throw `IllegalStateException` if there isn't enough space to insert an element

```
if(size == data.length) {
   throw new IllegalStateException ("The list is full");
}
```

**StaticArrayList.java**

# A Simple Array-Based Implementation (cont'd.)

- Good things:
  - Fast, random access of elements
  - Very memory efficient, very little memory is required other than that needed to store the elements
- Bad things:
  - Slow deletion and insertion of elements
  - Size must be known when the array is created and is fixed (static)

# A Simple Array-Based Implementation (cont'd.)

- Implement an array list with a dynamic array:
  - Let `push`($o$) be the operation that adds element $o$ at the end of the list
  - When the array is full, we replace the array with a larger one
  - How large should the new array be?
    - Incremental strategy: increase the size by a constant $c$
    - Doubling strategy: double the size

```
algorithm push(o)
    Input S
    Output S
    if t = S.length – 1 then
        A ← new array of size …
        for i ← 0 to n – 1 do
            A[i] ← S[i]
            S ← A
        n ← n + 1
        S[n-1] ← o
    return S
```

**`ArrayList.java`**

# Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of $n$ push operations
- We assume that we start with an empty list represented by a growable array of size 1
- We call amortized time of a push operation the average time taken by a push operation over the series of operations, i.e., $T(n)/n$

# Incremental Strategy Analysis

- Over $n$ push operations, we replace the array $k = n/c$ times, where $c$ is a constant
- The total time $T(n)$ of a series of $n$ push operations is proportional to

$$c + 2c + 3c + 4c + \ldots + kc =$$

$$c(1 + 2 + 3 + \ldots + k) =$$

$$ck(k + 1)/2$$

- Since $c$ is a constant, $T(n)$ is $O(k^2)$, i.e., $O(n^2)$
- Thus, the amortized time of a push operation is $O(n)$

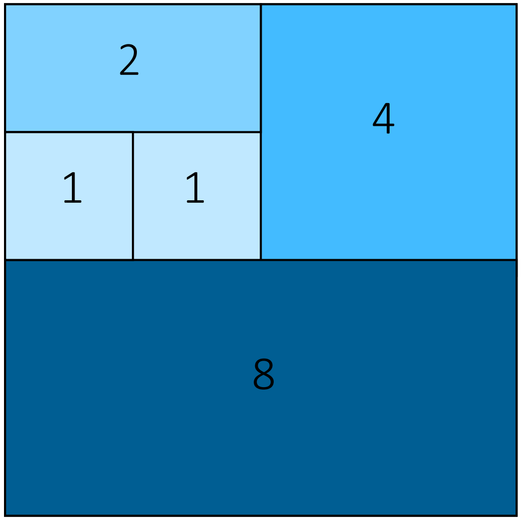
# Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of $n$ push operations is proportional to

$$1 + 2 + 4 + 8 + \ldots + 2^k = 2^{k+1} - 1 = 3n - 1$$

- $T(n)$ is $O(n)$
- The amortized time of a push operation is $O(1)$

geometric series



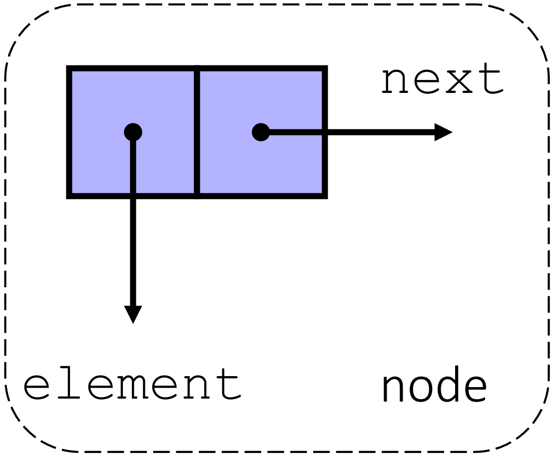
# Exercise 4.1

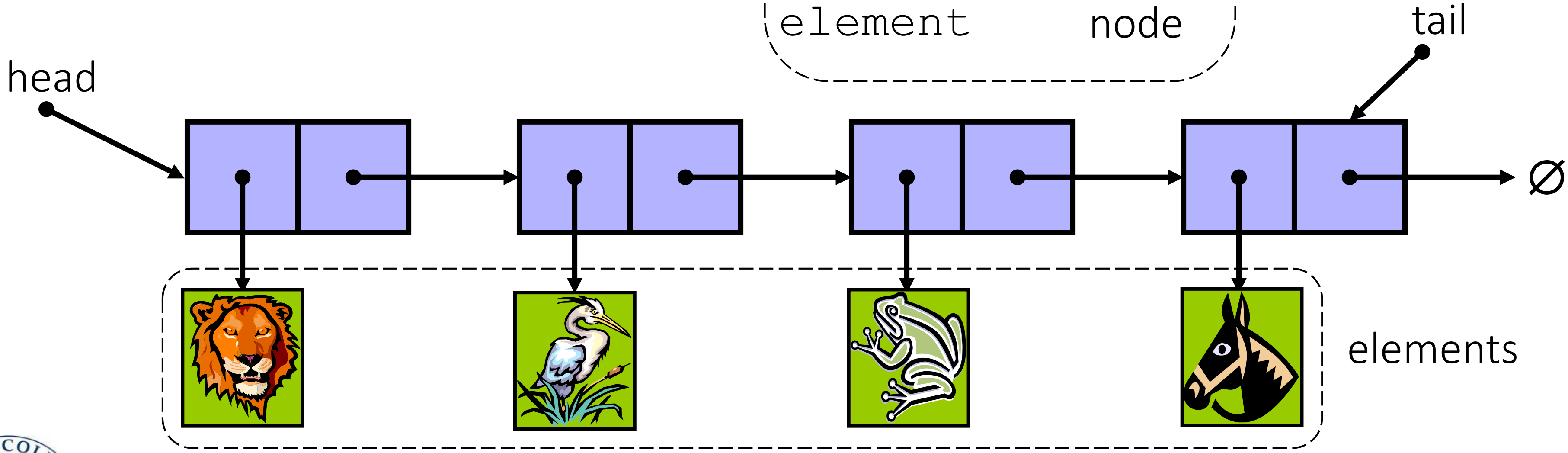
- Write methods `get(i)` and `set(i,e)`
- If index is out of range, throw `IndexOutOfBoundsException`

**`ArrayList.java`**

# 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



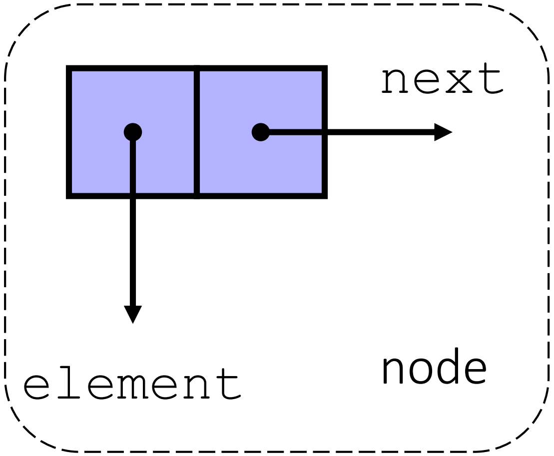
# Nodes

```java
public class Node<T> {
    public T element;
    public Node<T> next; // reference to next node

    /* Constructors */
    public Node() {
        next = null;
    }

    public Node(Node<T> n) {
        next = n;
    }

    public Node(T e, Node<T> n) {
        element = e;
        next = n;}
}
```




**Node.java**

# Operations

- Access/Update
- Search
- Insertion
  - Insert at the head
  - Insert at the tail
- Deletion
  - Delete at the head
  - Delete at the tail

`SinglyLinkedList.java`

| Method | Description |
| --- | --- |
| `size()` | Returns the number of elements in the list |
| `isEmpty()` | Returns a boolean indicating whether the list is empty |
| `first()` | Returns the first element in the list |
| `last()` | Returns the last element in the list |
| `addFirst(e)` | Adds a new element *e* to the front of the list |
| `addLast(e)` | Adds a new element *e* to the end of the list |
| `removeFirst()` | Removes and returns the first element of the list |
| `removeLast()` | Removes and returns the last element of the list |

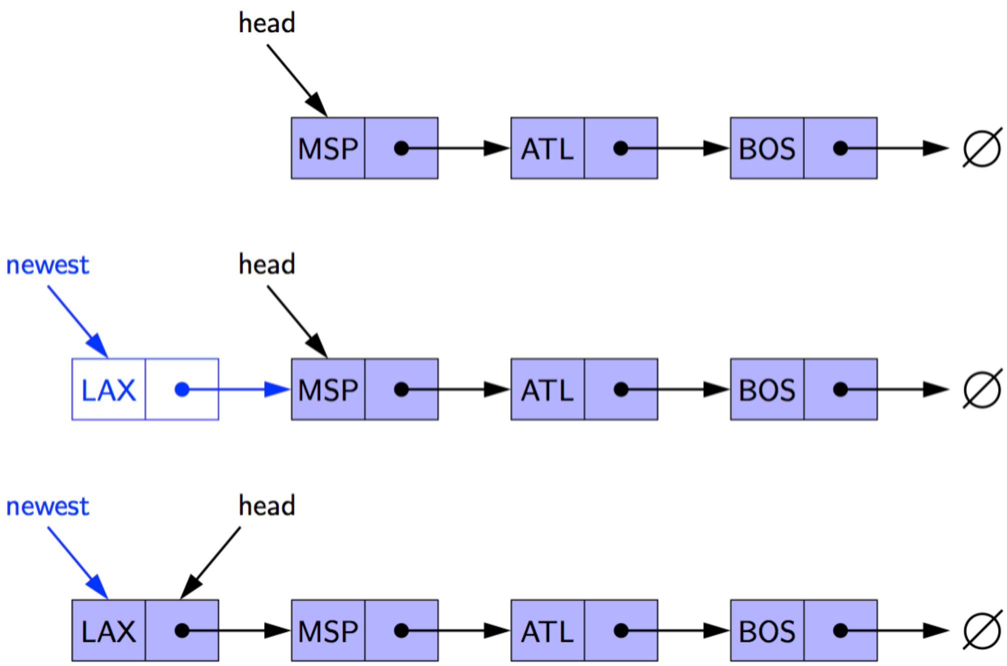
# Get Access to an Element

- Example: traverse the linked list to find the tail

```
/* traverse the linked list to find the tail */
Node<T> walk = head;
while(walk.next != null) {
   walk = walk.next;
}
```

# Inserting at the Head

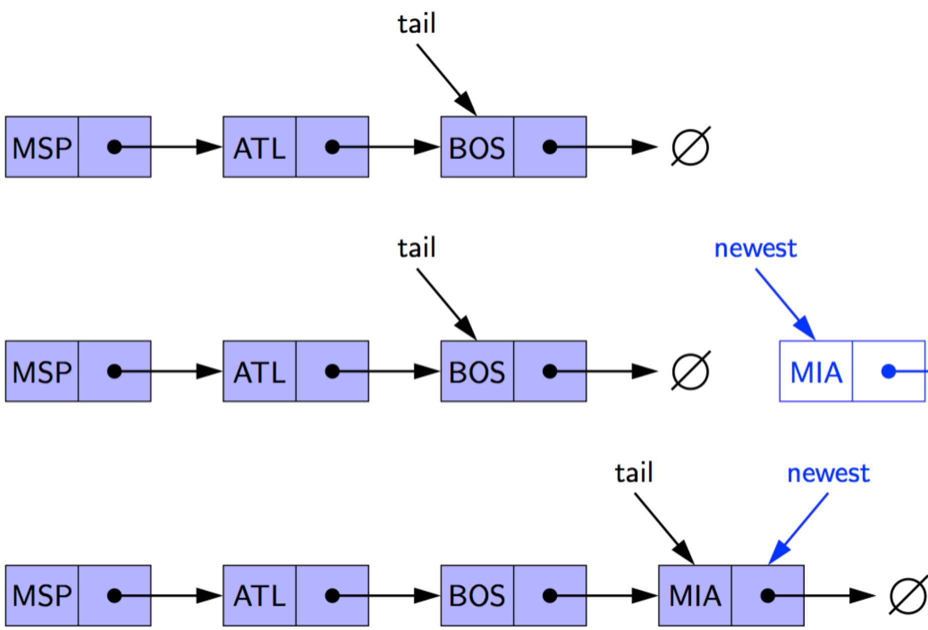
- Allocate a new node
- Insert the new element
- Have the new node point to the current `head`
- Update `head`, `size`




**SinglyLinkedList.java**

# Inserting at the Tail

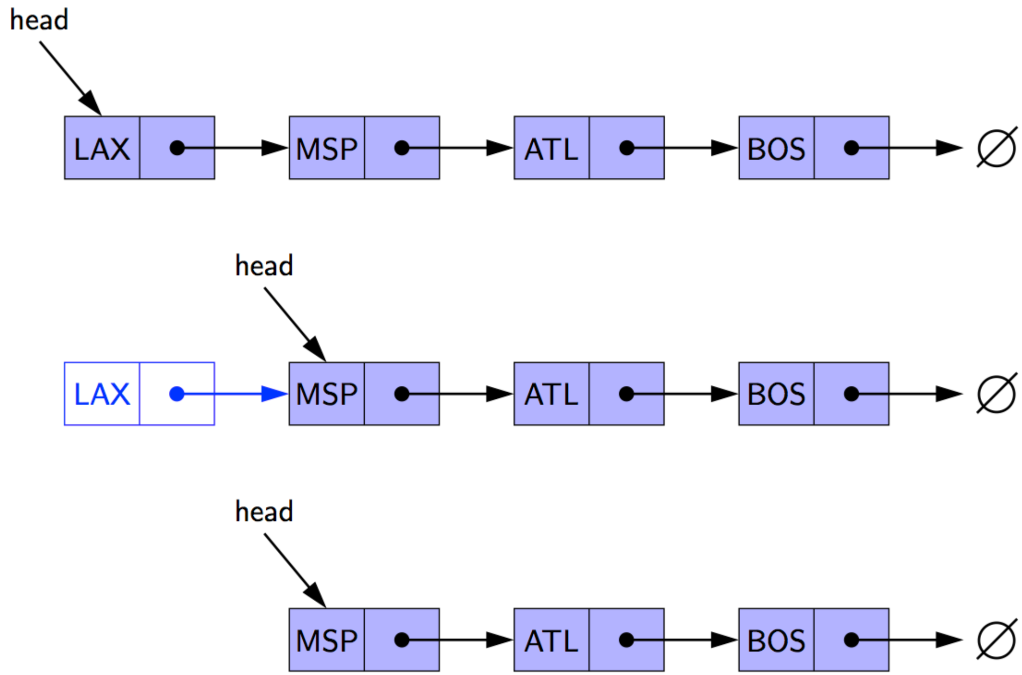
- Allocate a new node
- Insert the new element
- Have the new node point to `null`
- Have the current `tail` point to the new node
- Update `tail`, `size`




**SinglyLinkedList.java**

# Removing at the Head

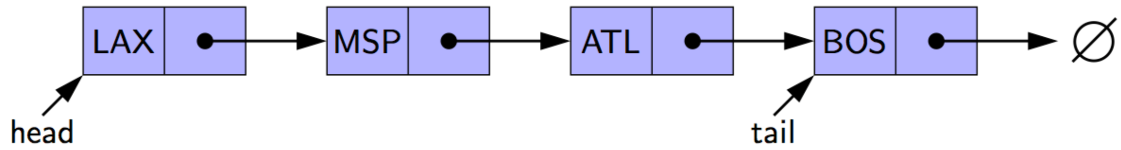
- Update `head` to point to the next node in the list
- Allow garbage collector to reclaim the former first node




**SinglyLinkedList.java**

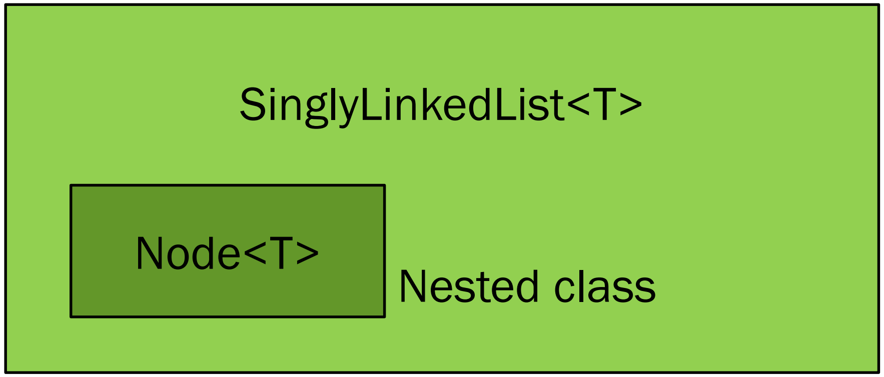
# Removing at the Tail

- Removing at the tail of a singly linked list is not efficient! Have to
- There is no constant-time way to update the `tail` to point to the previous node
- Finding the second last node → traversing the linked list → $O(n)$




**SinglyLinkedList.java**

# Implementation



# Performance

- Linked list is a dynamic data structure: the size can be altered at run time
- Inserting an item at the beginning of a linked list involves changing the new node's `next` field to point to the old `head`
- Deleting an item at the beginning of a linked list involves setting `head` to point to `head.next`

| | Access/Update | Search | Insertion | Deletion |
|---|---|---|---|---|
| Singly linked list | $O(n)$ | $O(n)$ | $O(1)$ | $\boxed{O(1)}$ |

Removing at the Tail: $O(n)$
How to improve: hash table!

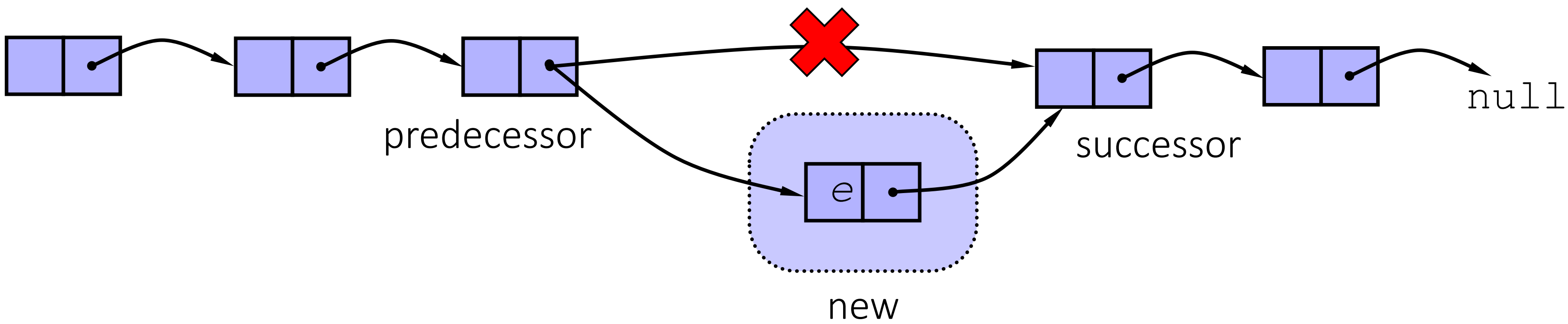
# Performance (cont'd.)

- To traverse a linked list, you start at head and then go from node to node, using each node's `next` field to find the next node
- A node with a specified key value can be found by traversing the list. Once found, an item can be displayed, deleted, or operated on in other ways
- A new link can be inserted before or after a link with a specified key value, following a traversal to find this node

| | Access/Update | Search | Insertion | Deletion |
|---|---|---|---|---|
| Singly linked list | $O(n)$ | $O(n)$ | $O(1)$ | $O(1)$ |

# Exercise 4.2: Inserting after a Given Node

- Insert element `e` into the linked list after a given node `predecessor`
- Assume the elements are unique

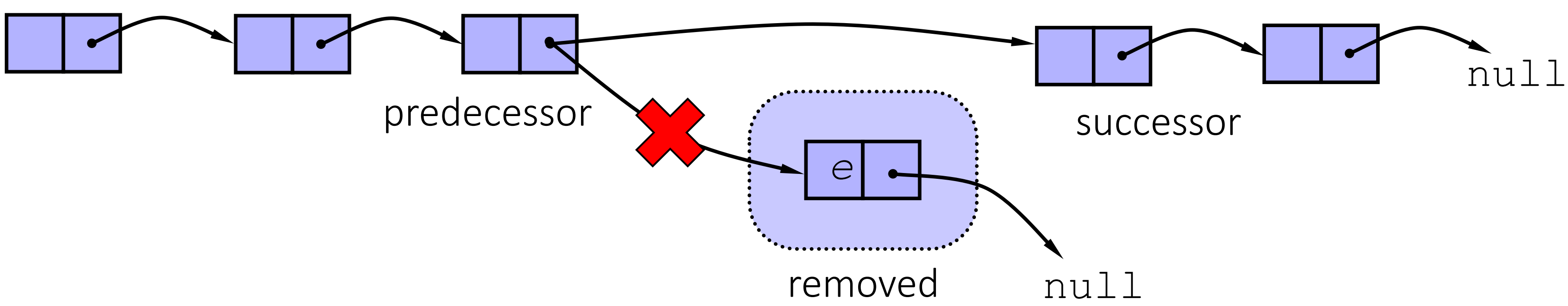



- `addBefore(T e, Node<T> n)`
  - `@param e, n`
- `addAfter(T e, Node<T> n)`
  - `@param e, n`

**SinglyLinkedList.java**

# Exercise 4.3: Removing after a Given Node

- Remove the given node from the list and return its element
- Assume the elements are unique

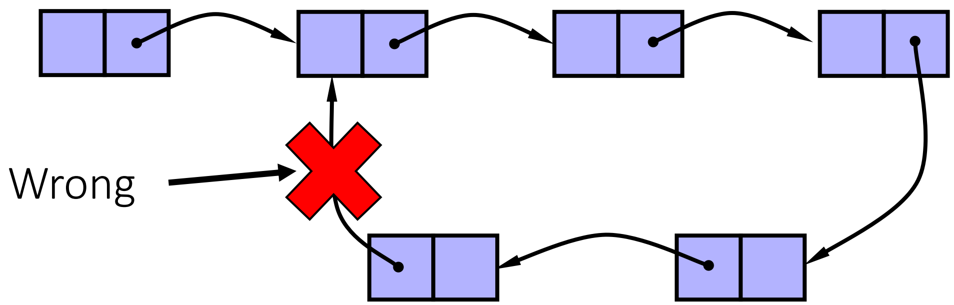



- `remove(Node<T> n)`
  - `@param n`
  - `@return e`

**`SinglyLinkedList.java`**

# Exercise 4.4: Finding Cycle in Linked List

- You are given a singly linked list. However, someone accidentally assigns the links wrong, which leads to a linked list with a loop. An example is shown in the following diagram:

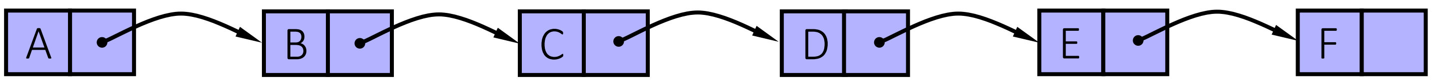



  - Find the loop
  - Correct the error by having the wrongfully linked node point to `null`

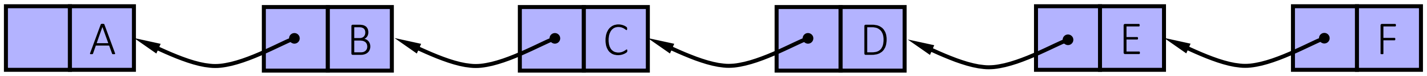
**`CycleInLinkedList.java`**

# Exercise 4.5: Reversing Linked Lists

- Given a singly linked list, reverse the order of elements
  - Before:




  - After:




`ReverseLinkedList.java`

# Array versus Linked Lists

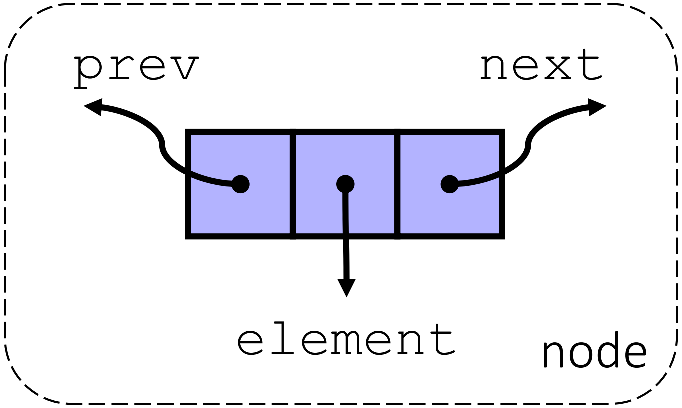
- Linked lists are more complex to code and manage than arrays, but they have some distinct advantages
  - Dynamic: a linked list can easily grow and shrink in size
    - We don't need to know how many nodes will be in the list. They are created in memory as needed
    - In contrast, the size of a C++ array is fixed at compilation time
  - Easy and fast insertions and deletions
    - To insert or delete an element in an array, we need to copy to temporary variables to make room for new elements or close the gap caused by deleted elements
    - With a linked list, no need to move other nodes. Only need to reset some pointers

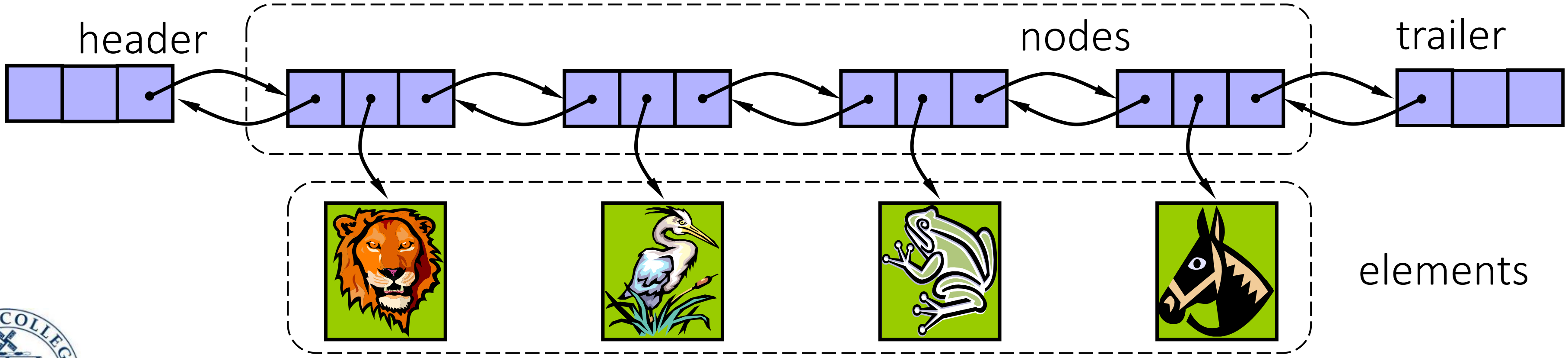
# Linked List

- Variants:
  - Doubly linked list
  - Circularly linked list

# 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:
  - Dummy nodes
  - Sentinels, or guards



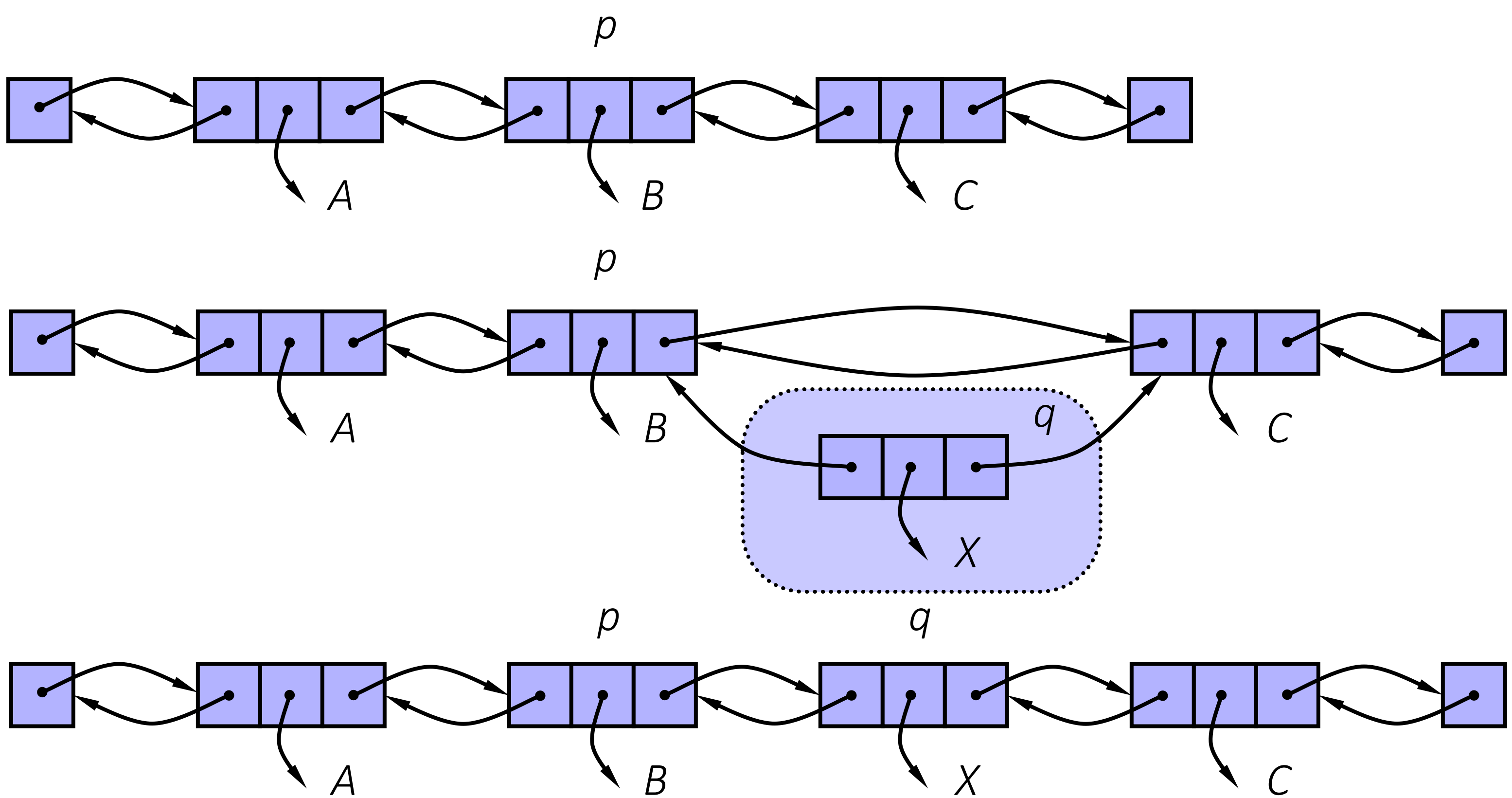



| Method | Description |
|---|---|
| `size()` | Returns the number of elements in the list |
| `isEmpty()` | Returns a boolean indicating whether the list is empty |
| `first()` | Returns the first element in the list |
| `last()` | Returns the last element in the list |
| `addFirst(e)` | Adds a new element *e* to the front of the list |
| `addLast(e)` | Adds a new element *e* to the end of the list |
| `removeFirst()` | Removes and returns the first element of the list |
| `removeLast()` | Removes and returns the last element of the list |

**`DoublyLinkedList.java`**

# Insertion

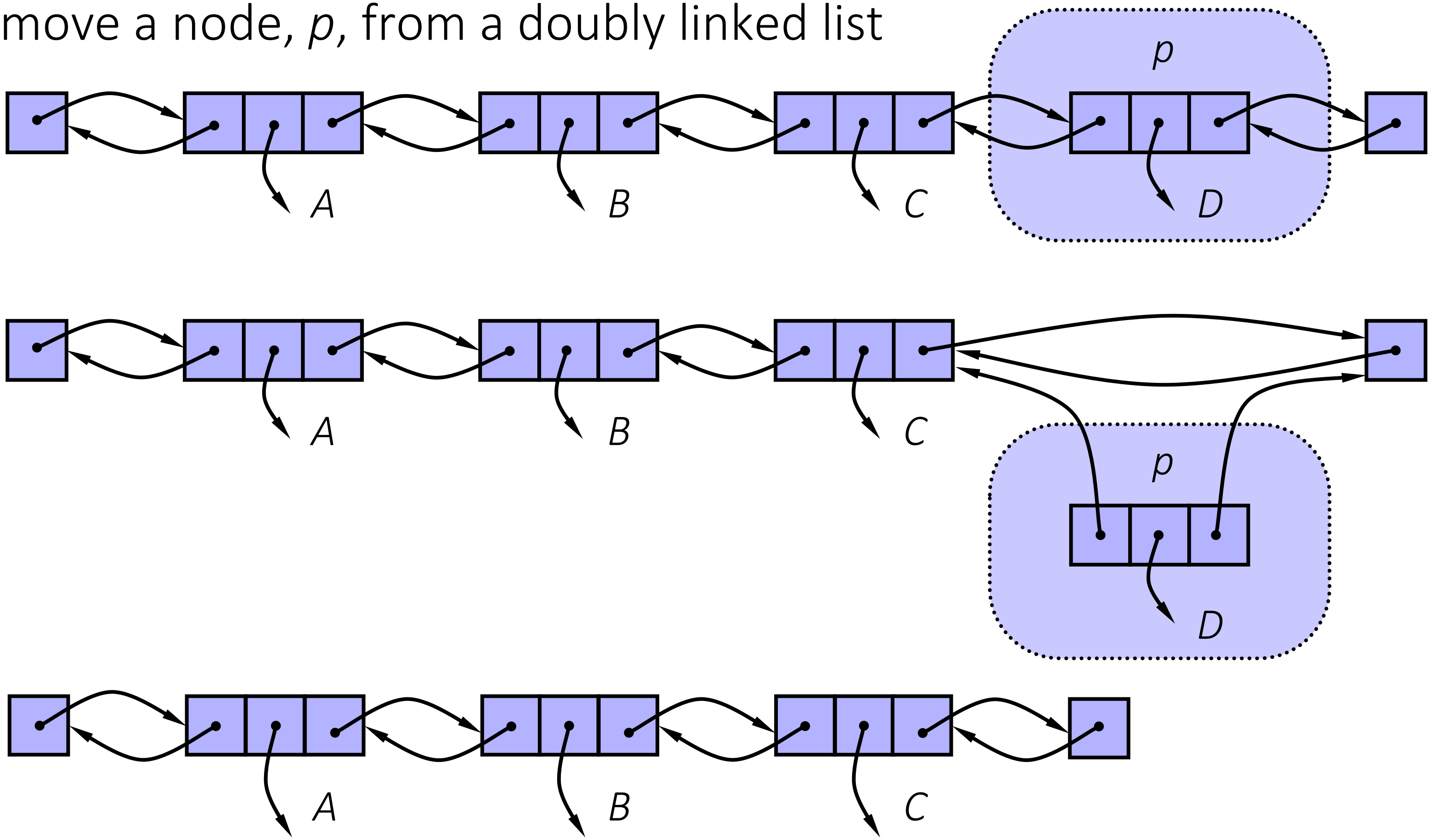
- Insert a new node, *q*, between *p* and its successor




DoublyLinkedList.java

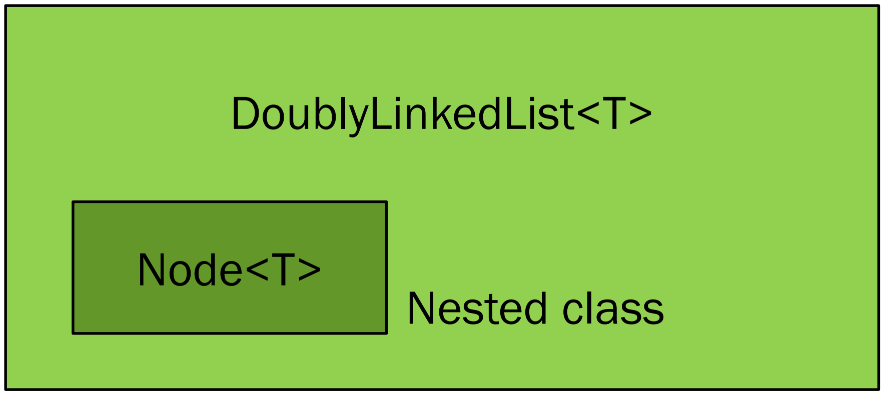
# Deletion

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




`DoublyLinkedList.java`

# Implementation



# Performance

- A doubly linked list maintains a pointer to the last node in the list, often called last, as well as to the first
- A doubly linked list allows insertion at the end of the list

| | Access/Update | Search | Insertion | Deletion |
|---|---|---|---|---|
| Doubly linked list | $O(n)$ | $O(n)$ | $O(1)$ | $O(1)$ |

# Exercise 4.6 : Inserting before/after a Node

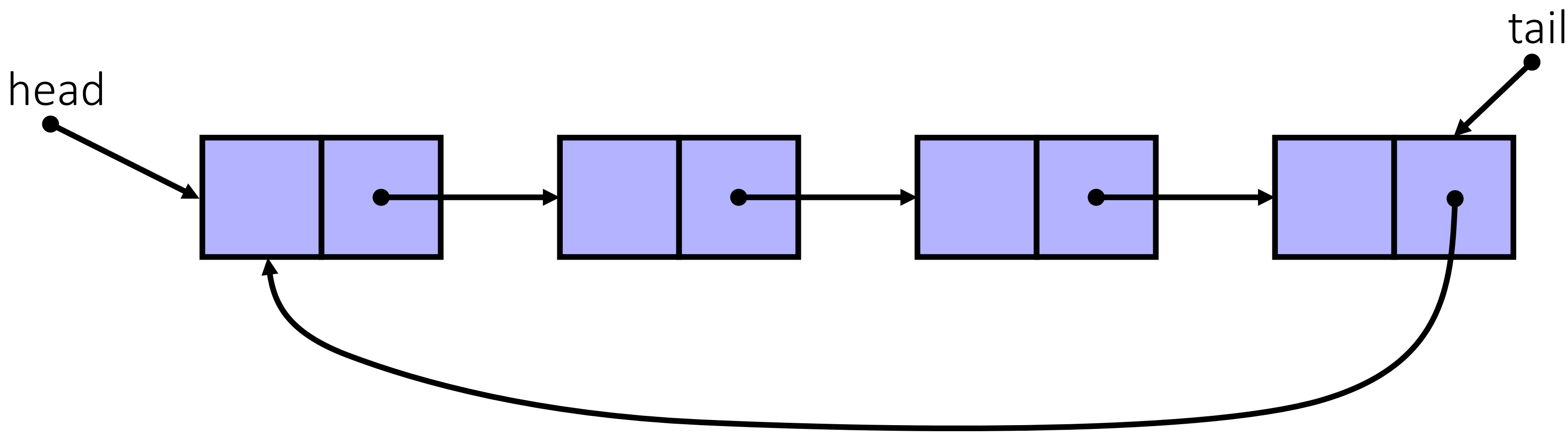
- Given a node, insert the new element $e$ before/after that node
- Assume the elements are unique
- `addBefore(T e, Node<T> n)`
  - `@param e, n`
- `addAfter(T e, Node<T> n)`
  - `@param e, n`

# Exercise 4.7: Removing after a Given Node

- Remove the given node from the list and return its element
- Assume the elements are unique
- `remove(Node<T> n)`
  - `@param n`
  - `@return e`

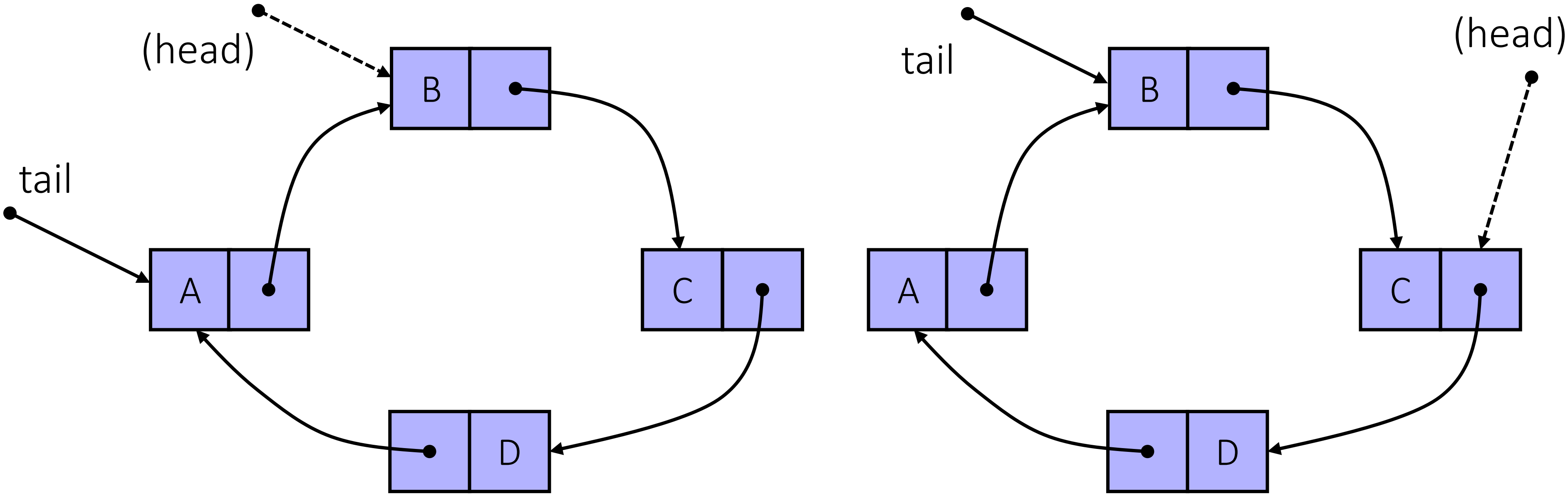
# Circularly Linked List

- A circularly linked list, which is essentially a singly linked list in which the `next` reference of the tail is set to refer back to the `head` of the list (rather than `null`)




**`CircularlyLinkedList.java`**

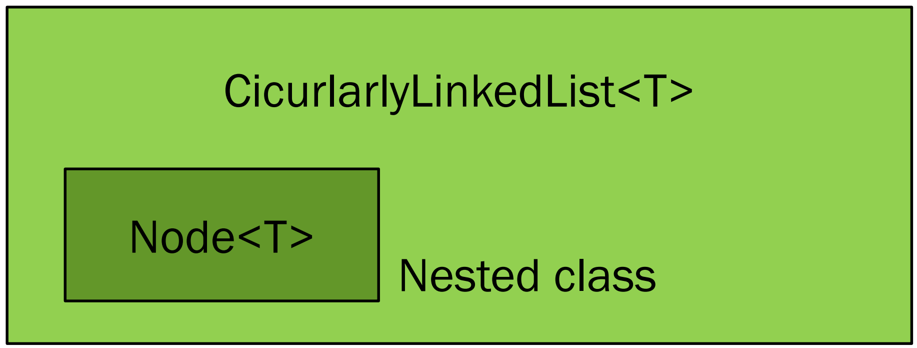
| Method | Description |
| --- | --- |
| `size()` | Returns the number of elements in the list |
| `isEmpty()` | Returns a boolean indicating whether the list is empty |
| `first()` | Returns the first element in the list |
| `last()` | Returns the last element in the list |
| `addFirst(e)` | Adds a new element *e* to the front of the list |
| `addLast(e)` | Adds a new element *e* to the end of the list |
| `removeFirst()` | Removes and returns the first element of the list |
| `removeLast()` | Removes and returns the last element of the list |
| `rotate()` | Moves the first element to the end of the list |




```java
/* rotate: moves the first element to the end of the list */
public void rotate () {
    if (tail != null) {  // if empty, do nothing
        tail = tail.next; // the old head becomes the new tail
    }
}
```

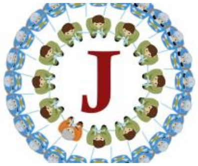
`CircularlyLinkedList.java`

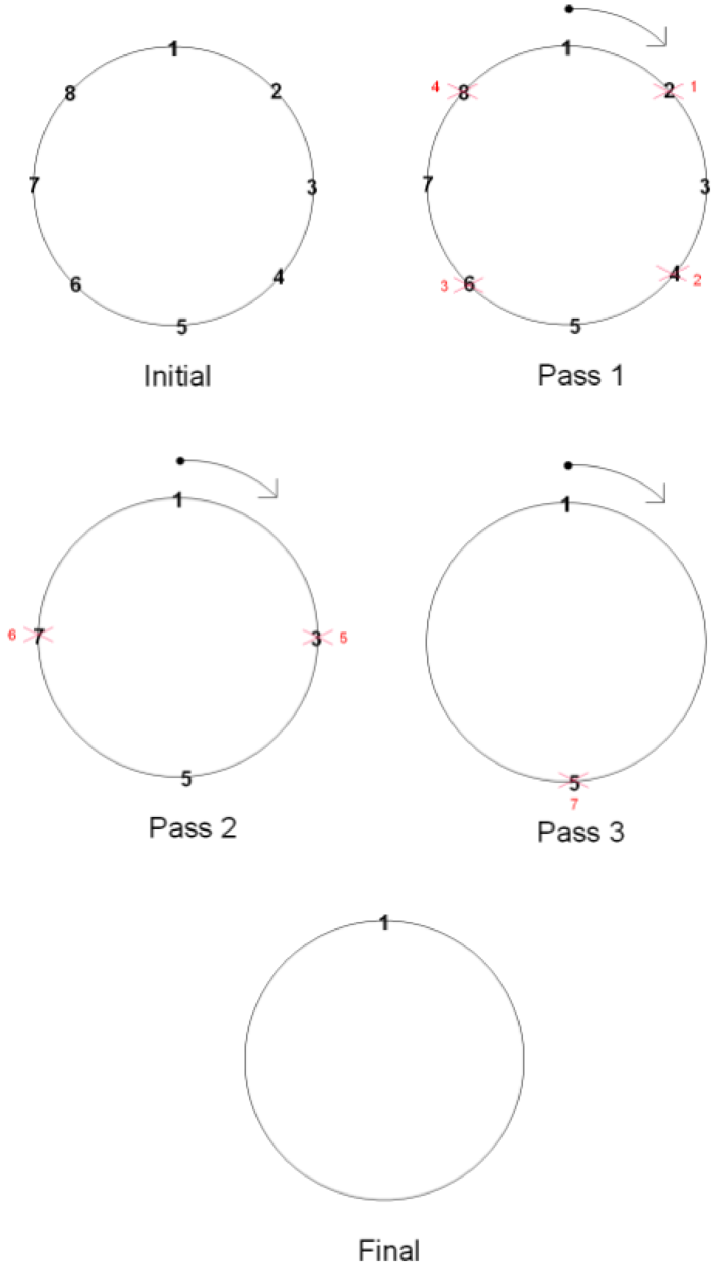
# Implementation



# Example: Josephus problem

- The **Josephus problem** (or **Josephus permutation**) is a theoretical problem related to a certain counting-out game
  - People are standing in a circle waiting to be executed. The counting begins at a specified point in the circle and proceeds around the circle in a fixed direction. After a specified number of people are skipped, the next person is executed. The procedure is repeated with the remaining people, starting with the next person, going in the same direction and skipping the same number of people, until only one person remains, and is freed. Given the total number of people *n*, and a number *k* which indicates that *k*-1 persons are skipped and the *k*th person is executed, the task is to write a program that can choose the position in the initial circle to avoid execution





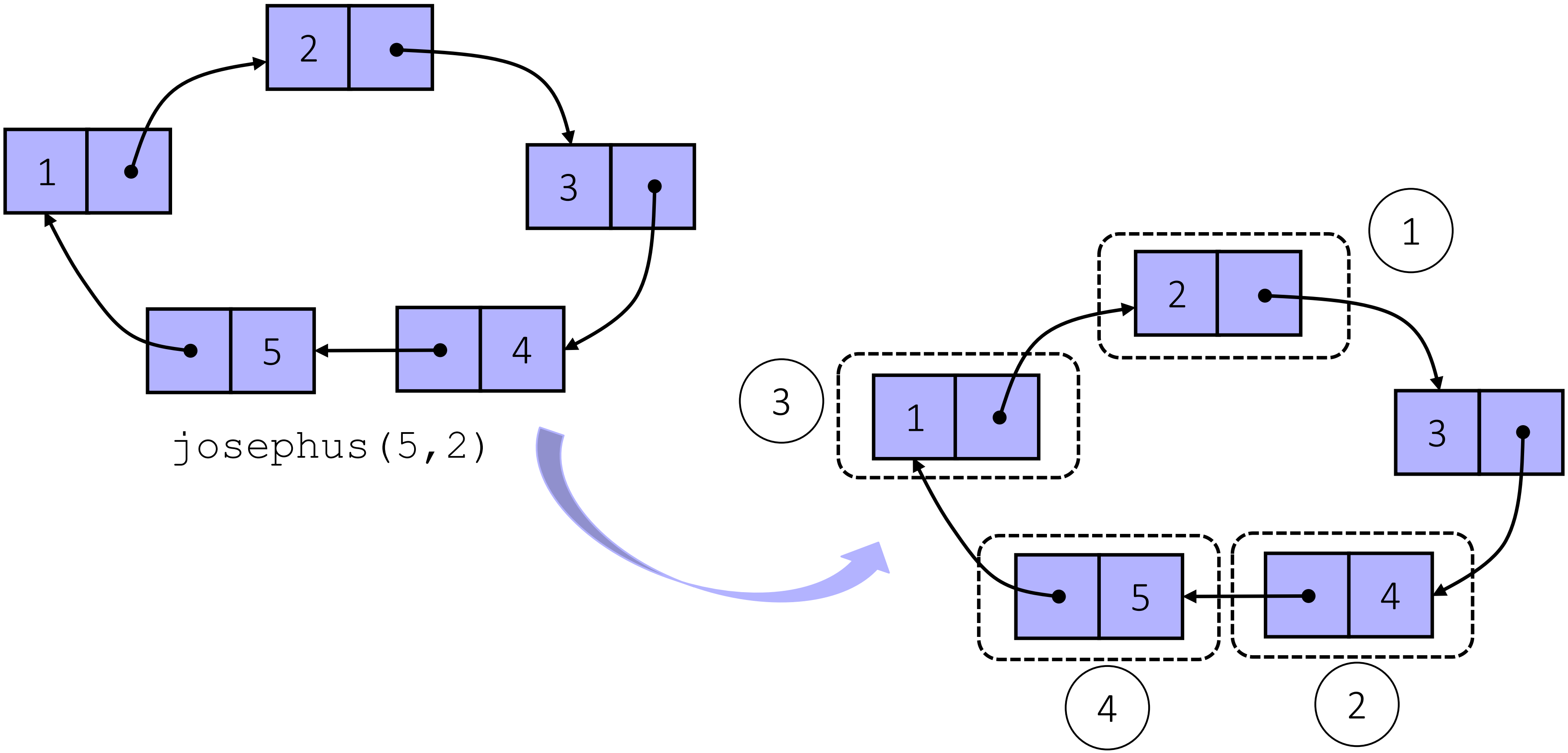

- Example:
  - `josephus(`$n$`,`$k$`)`
  - $n$=8, $k$=2
- Exercise 4.8:
  - `josephus(41,2)`

# Recursive Solution

- Recursive solution:
  - `josephus(`$n$`,`$k$`) = (josephus(`$n$`-1,`$k$`)+`$k$`-1)%`$n$` + 1`
  - `josephus(1,`$k$`) = 1`

**RecursiveJosephus.java**

# Linked List-based Solution




`LinkedListJosephus.java`

# Comparison of Solutions

| Methods | Time complexity |
|---|---|
| Recursion | $O(n)$ |
| Linked list | $O(n)$ |
| Array | $O(n^2)$ |

# Summary

- Abstract data type (ADT) : specify data and operations
- List ADT (Array list):
  - Data, operations
  - Array-based implementation
- Linked list ADT:
  - Singly linked list:
    - Data, operations
    - Implementation
  - Variants:
    - Doubly linked list
    - Circular linked list
      - Josephus problem