Data Structures & Algorithms

Adil M. Khan

Professor of Computer Science

Innopolis University

a.khan@innopolis.ru

Recap

- What is algorithm analysis?
- Why do we analyze algorithms?
- How do we analyze algorithms?

Objectives of Today's Lecture

- Abstract Data Type vs. Data Structure
- Basic building blocks of any data structure
- List ADT: its implementations and analysis
- Stack ADT: its implementations and analysis
- Queue ADT: its implementations and analysis

Disclaimer!!

- This will be an easy but long and fast lecture – stay attentive
- I will cover analysis of some important operations – others will be left for you as exercises – learn the concepts and then apply

Data Structure

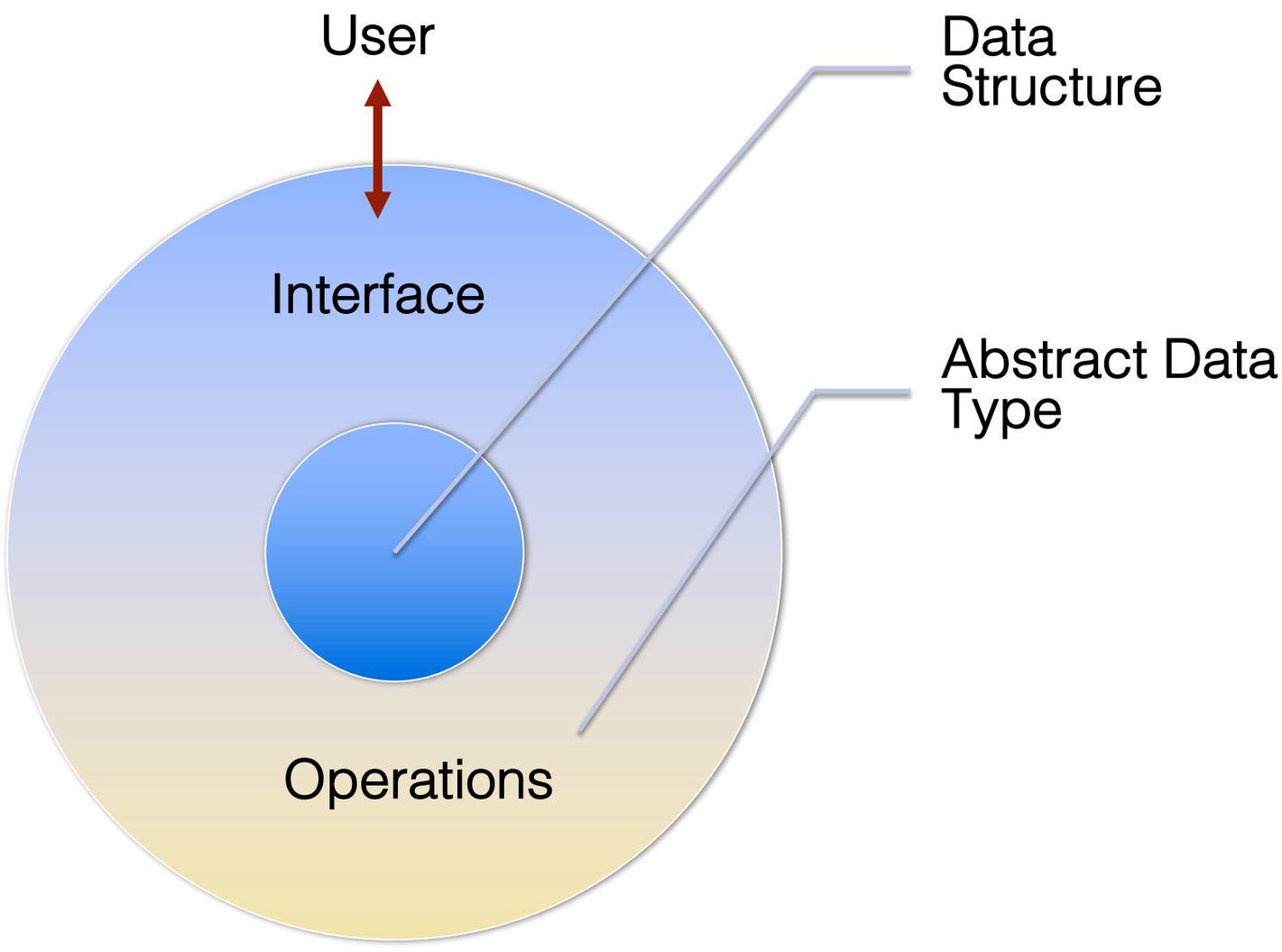
- The physical implementation of
  - how the data will be stored
  - And how the data will be manipulated

Abstract Data Type

- Description of how we view the data and the operations that are allowed without regard to how they will be implemented.

- This means that we are concerned only with what the data structure is representing and not with how it will eventually be constructed

ADT vs. DS


User
Interface
Operations
Data Structure
Abstract Data Type

Two Basic Building Blocks of a Data Structure

- Contiguously Allocated Data Structures [Arrays]
- Linked Data Structures [Linked Nodes]

List

Applications

- Attendance monitoring system (List of attendees)
- Computer Games (List of top scores)
- Online shopping (List of selected items)
- …

List as an ADT

- Arrangement of Data:
  - Completely unrestricted sequence of zero or more items
- Operations
  - add, update, and remove items at any position
  - lookup an item by position
  - find the index at which a given value appears
  - Is the list empty
  - …

List ADT

```
1  /** A simplified version of the java.util.List interface. */
2  public interface List<E> {
3    /** Returns the number of elements in this list. */
4    int size( );
5
6    /** Returns whether the list is empty. */
7    boolean isEmpty( );
8
9    /** Returns (but does not remove) the element at index i. */
10   E get(int i) throws IndexOutOfBoundsException;
11
12   /** Replaces the element at index i with e, and returns the replaced element. */
13   E set(int i, E e) throws IndexOutOfBoundsException;
14
15   /** Inserts element e to be at index i, shifting all subsequent elements later. */
16   void add(int i, E e) throws IndexOutOfBoundsException;
17
18   /** Removes/returns the element at index i, shifting subsequent elements earlier. */
19   E remove(int i) throws IndexOutOfBoundsException;
20 }
```

Code Fragment 7.1: A simple version of the List interface.

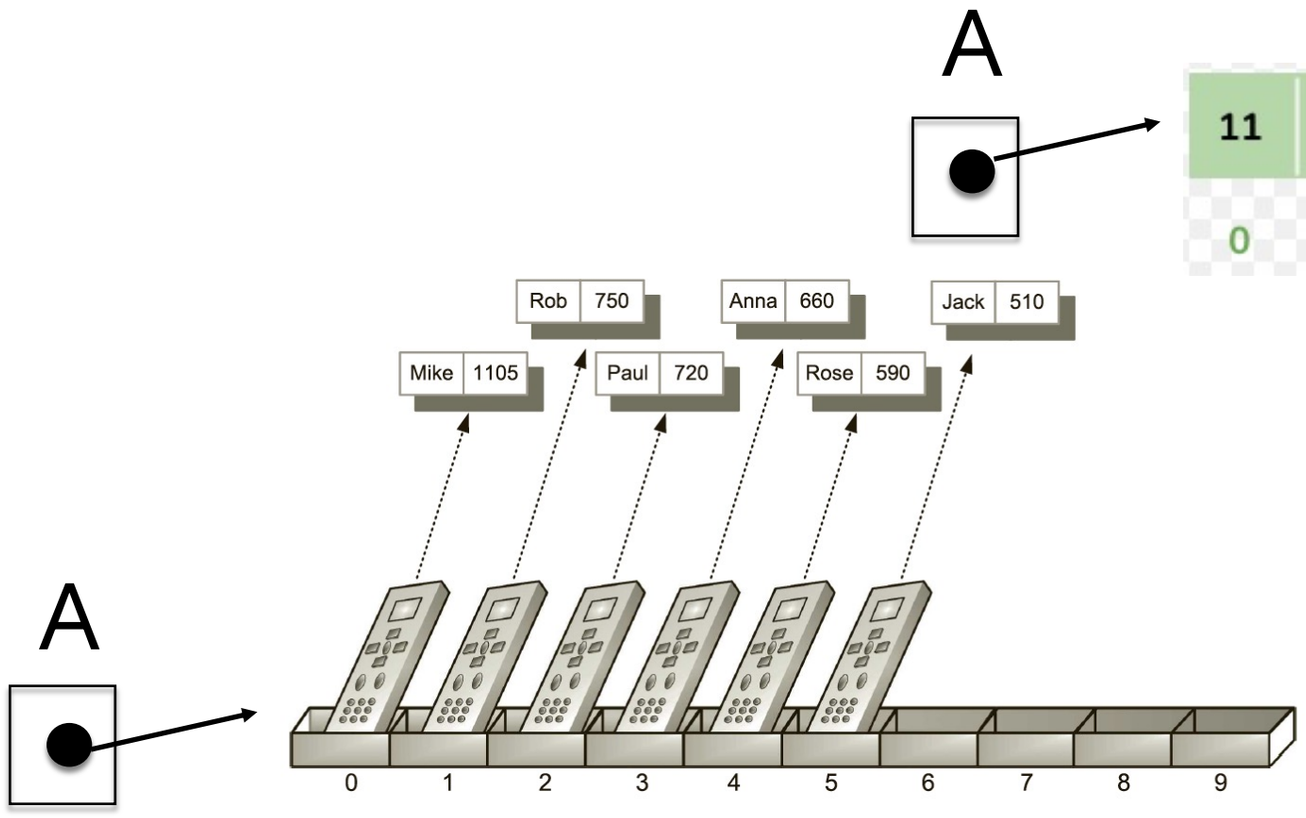
The key idea is that we have not specified how the list is to be implemented

List ADT Implementation

- **Array-List**
  - uses array
  - reasonable if we know in advance the maximum number of the items in the list
- Linked-List
  - uses linked nodes
  - best if we don't know in advance the number of elements in the list (or if it varies greatly)

Array-based Lists

- Implementation of the list ADT using an array A
- A[i] stores a reference to the element (or the element itself) at index i



ArrayList

```
1  public class ArrayList<E> implements List<E> {
2    // instance variables
3    public static final int CAPACITY=16;      // default array capacity
4    private E[ ] data;                        // generic array used for storage
5    private int size = 0;                     // current number of elements
6    // constructors
7    public ArrayList( ) { this(CAPACITY); }   // constructs list with default capacity
8    public ArrayList(int capacity) {          // constructs list with given capacity
9      data = (E[ ]) new Object[capacity];     // safe cast; compiler may give warning
10   }
```

Code Fragment 7.2: An implementation of a simple ArrayList class with bounded capacity. (Continues in Code Fragment 7.3.)

For complete code, refer to Goodrich's book, Ch: 7

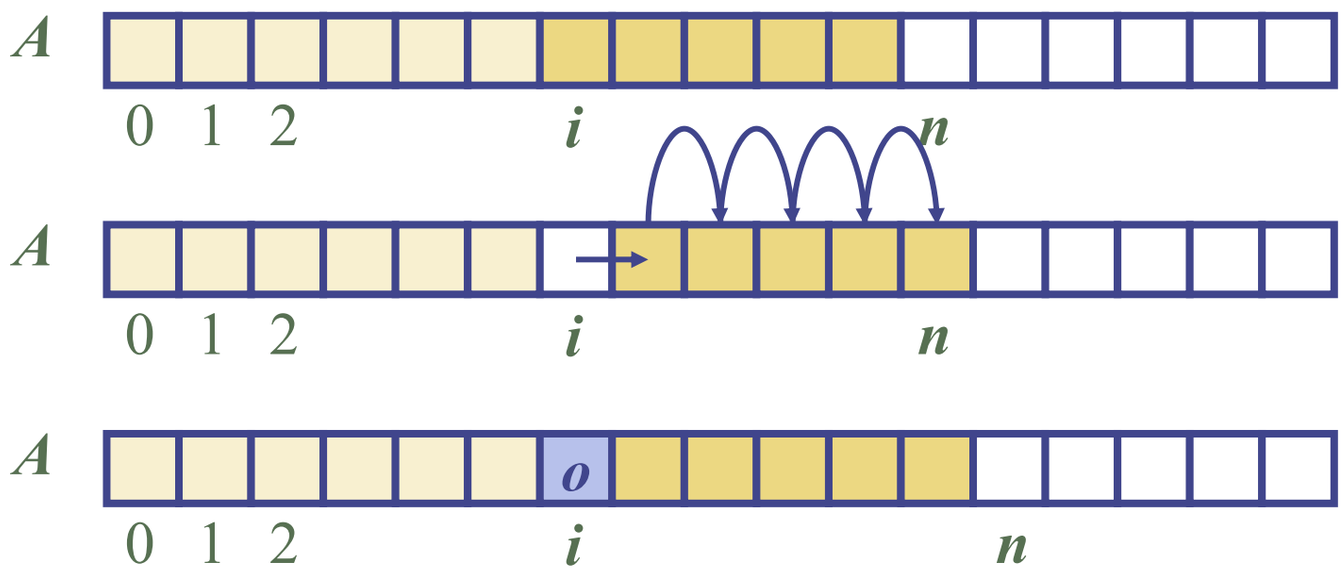
Analysis

- set(i,e) and get(i)
  - Easy to implement
  - Just access A[i] (assuming i is a legitimate index)
- What are their Time complexity?

```
17    public E get(int i) throws IndexOutOfBoundsException {
18      checkIndex(i, size);
19      return data[i];
20    }
21    /** Replaces the element at index i with e, and returns the replaced element. */
22    public E set(int i, E e) throws IndexOutOfBoundsException {
23      checkIndex(i, size);
24      E temp = data[i];
25      data[i] = e;
26      return temp;
27    }
```

Analysis

- add(i,e)

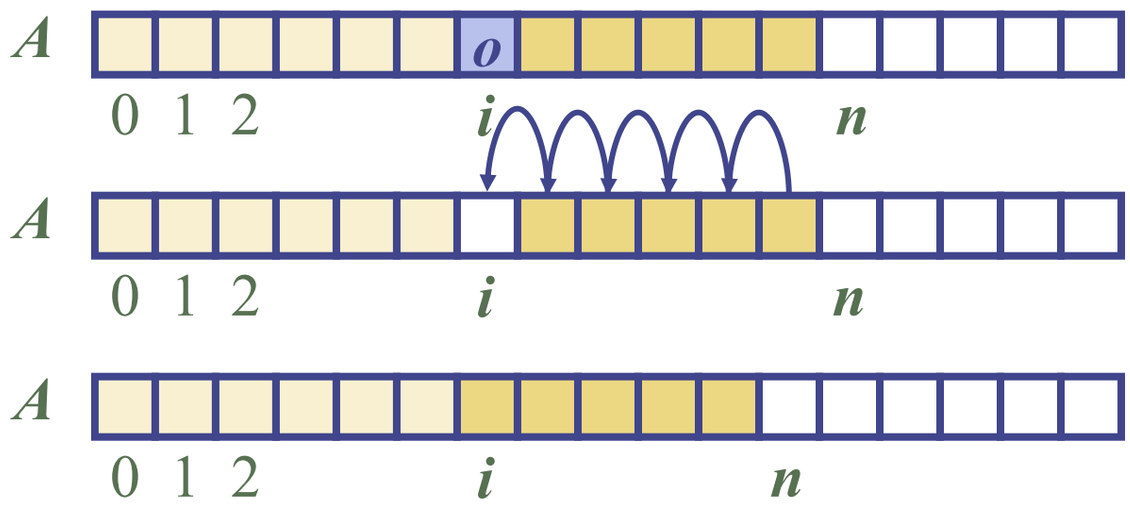


- Shifting items forward to create space if index i is occupied

- Time complexity?

Analysis

- remove(i)



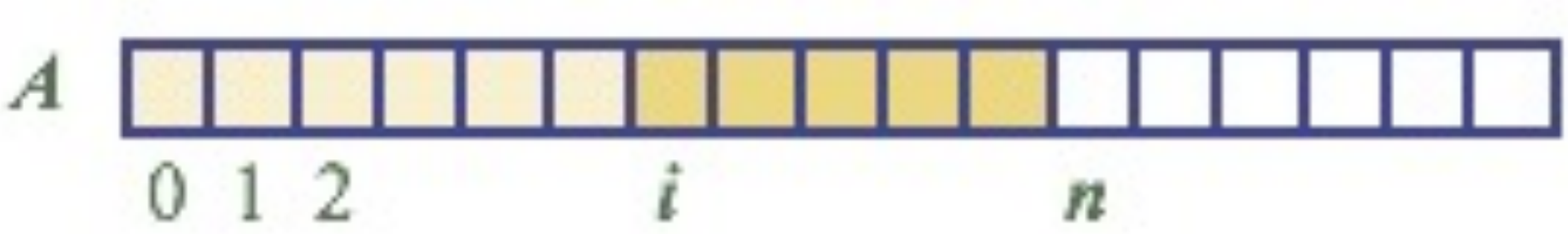
- Shifting items backwards to fill space
- Time complexity?

Analysis-Summary

Method	Running Time
size()	O(1)
isEmpty()	O(1)
get(i)	O(1)
set(i, e)	O(1)
add(i, e)	O(n)
remove(i)	O(n)

Performance of an array list with n elements realized by a fixed-capacity array.

What About This?



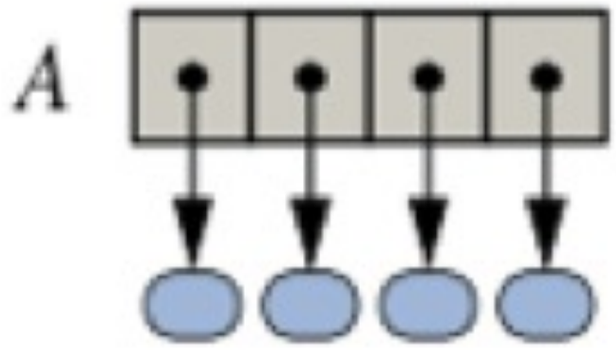
- n represents the first empty index in the list,

- Let's suppose we have a method *addLast(e)* that adds a new item at the end of the sequence – at index n

- Time Complexity?

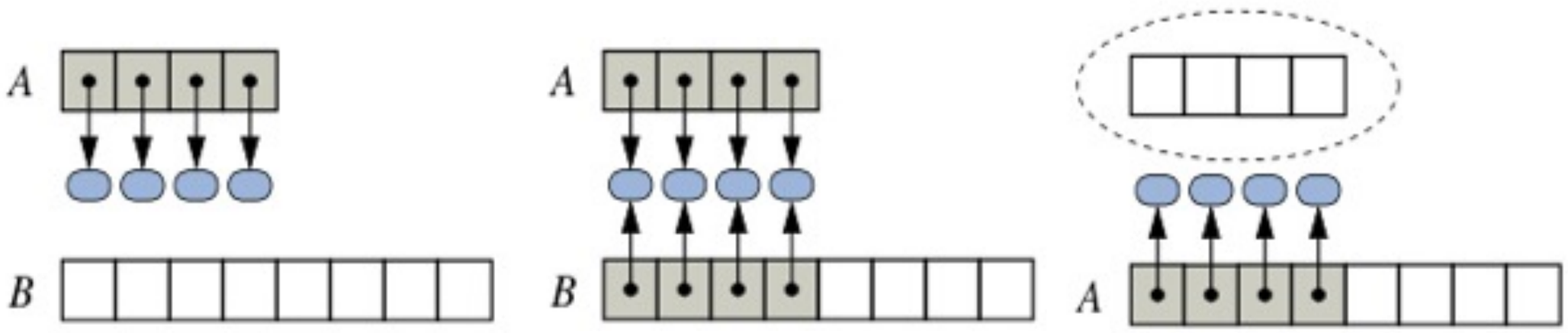
But What if the List is Full?

- addLast(e)



Dynamic Arrays-based Lists

- When the array is full, we replace the array with a larger one



An illustration of "growing" a dynamic array: (1) create new array *B*; (2) store elements of *A* in *B*; (3) reassign reference *A* to the new array.

Time Complexity?

Dynamic Arrays-based Lists

- How large should the new array be?
  - ***Doubling strategy:*** start with a list of size 1, and then double the size every time the list is full

Dynamic Arrays-based Lists

- Doubling Strategy – Amortized Analysis

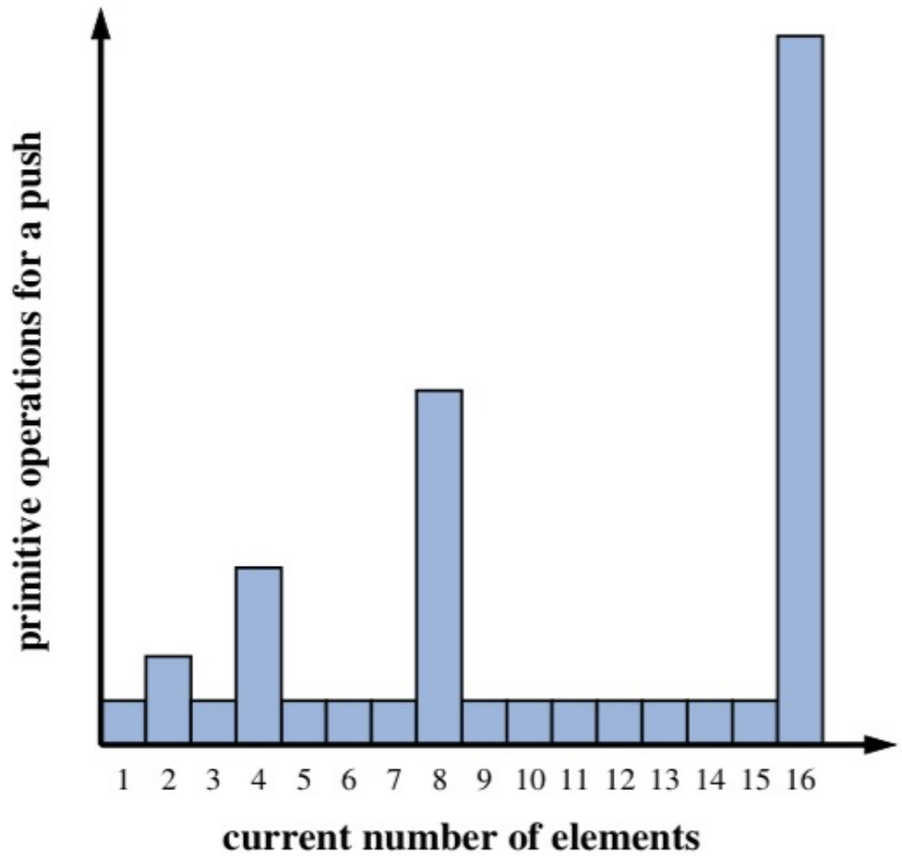


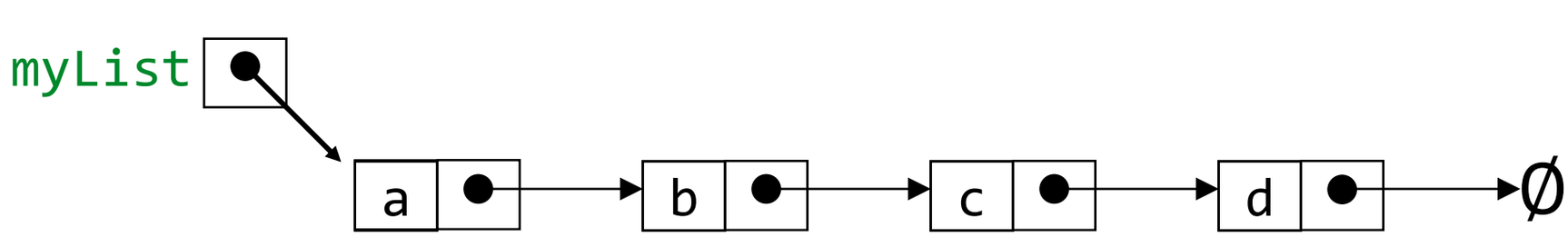
Figure 7.4: Running times of a series of push operations on a dynamic array.

You will learn more about Amortized Analysis in Tutorial

Singly Linked List

Anatomy of a singly linked list

- A singly linked list consists of a sequence of nodes



Each node contains a value
and a link (pointer or reference) to some other node
The last node contains a null link
And lastly the reference to the list object

Terminology-1

- A node's successor is the next node in the sequence
  - The last node has no successor
- A node's predecessor is the previous node in the sequence
  - The first node has no predecessor
- A list's length is the number of elements in it
  - A list may be empty (contain no elements)

Terminology-2

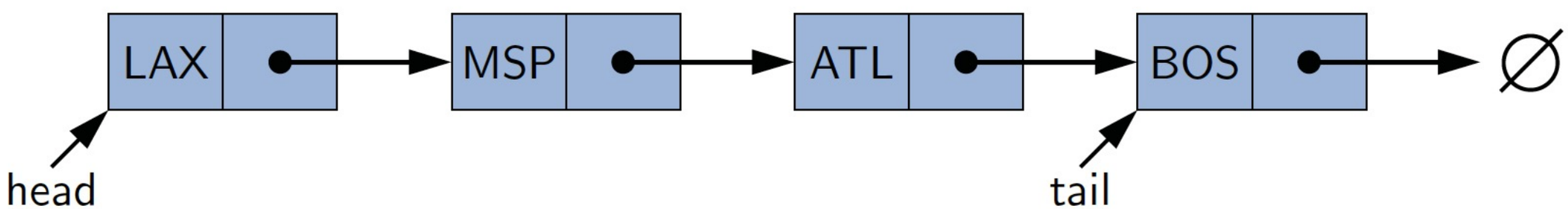
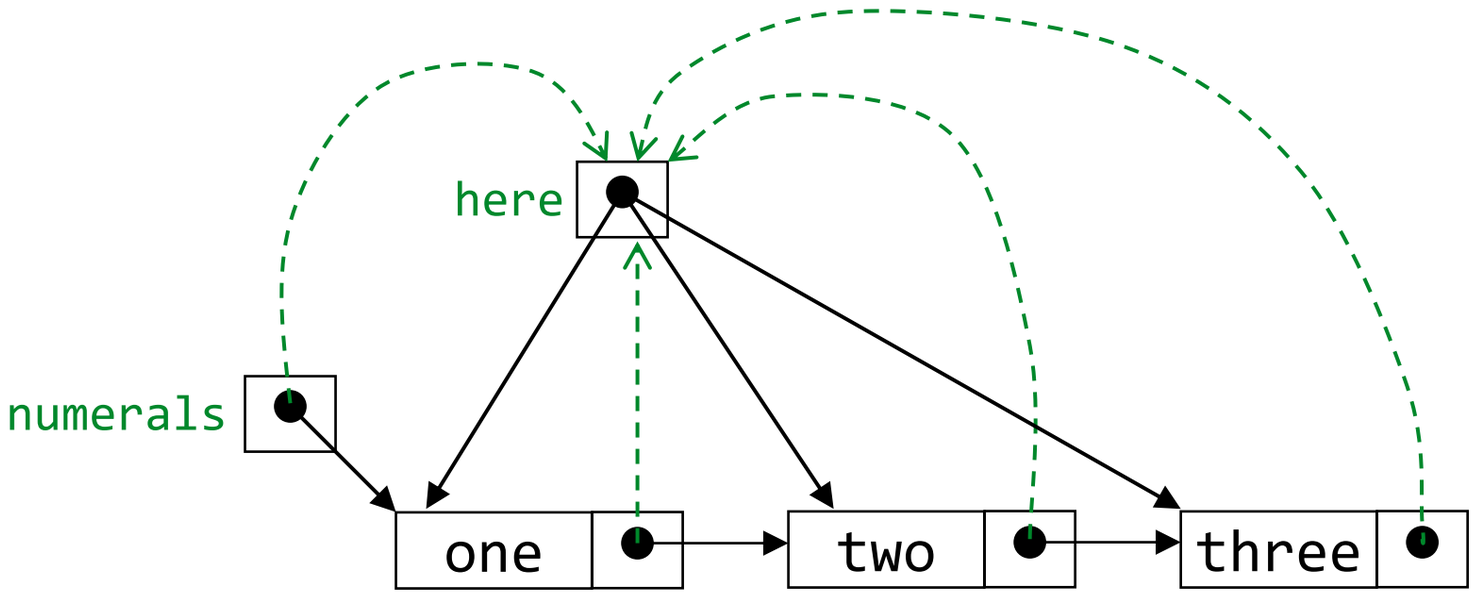


Figure 3.11: Example of a singly linked list whose elements are strings indicating airport codes. The list instance maintains a member named head that refers to the first node of the list, and another member named tail that refers to the last node of the list. The **null** value is denoted as Ø.

- List hopping or traversing starting at the head, and moving from one node to another by following each node's next pointer

Traversing a SLL



Analysis

- set(i,e) and get(i)

  1. You will have to arrive at the i-th node first
  2. Then simply set or get its value

Analysis

- set(i,e) and get(i)

  1. You will have to arrive at the i-th node first
  2. Then simply set or get its value

- Time complexity?

Analysis

- add(i,e)

  1. You will have to arrive at (i -1)-th node
  2. Insert the item at i-th position and update the links

Analysis

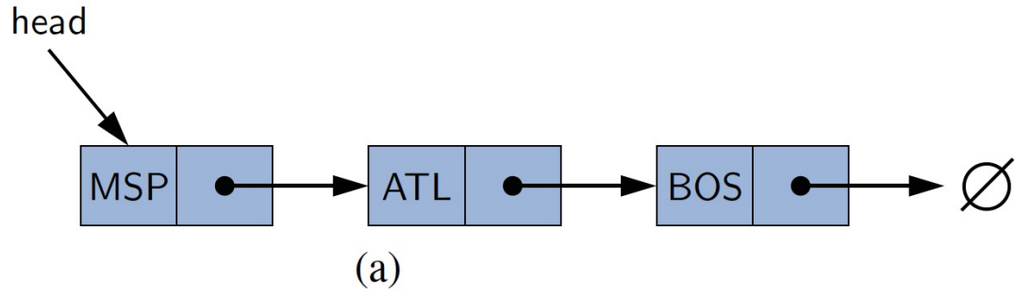
- add(i,e)

1. You will have to arrive at (i -1)-th node
2. Insert the item at i-th position and update the links

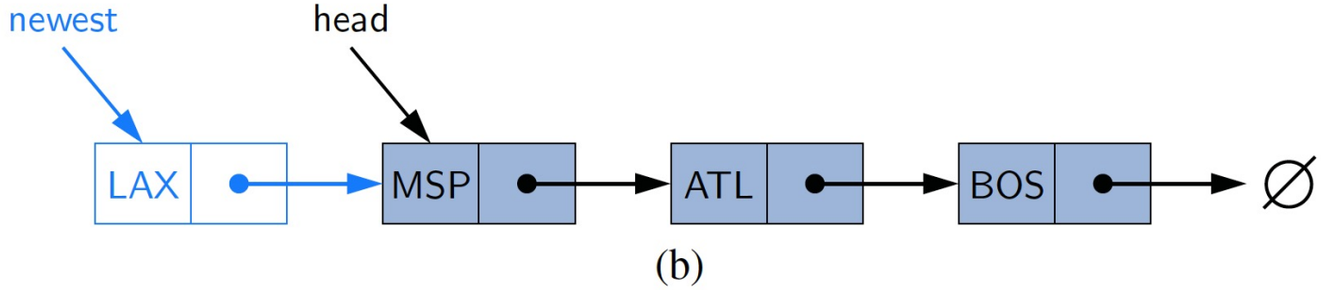
- Time complexity?

Inserting At Front

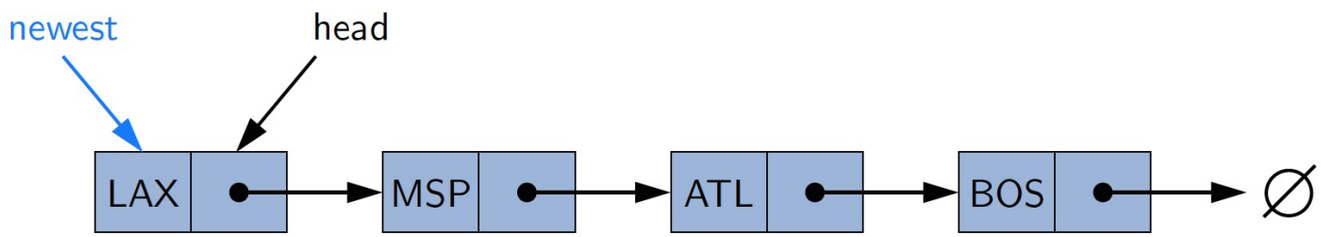
- addFirst(e)
- It works as shown here
- Time complexity?



(a)



(b)



(c)

Inserting at the End

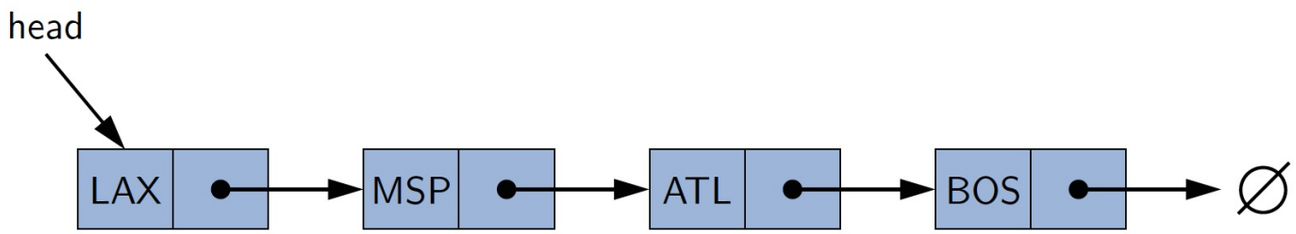
- addLast(e)
- How will you implement this method?
  - Do it for both (a) when we have the tail pointer, and (b) when we do not have the tail pointer
- Time complexity?

Removing the First Element

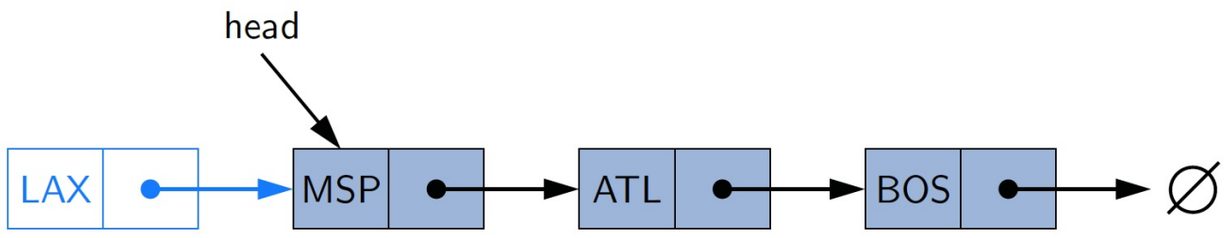
- removeFirst()

- It works as shown here

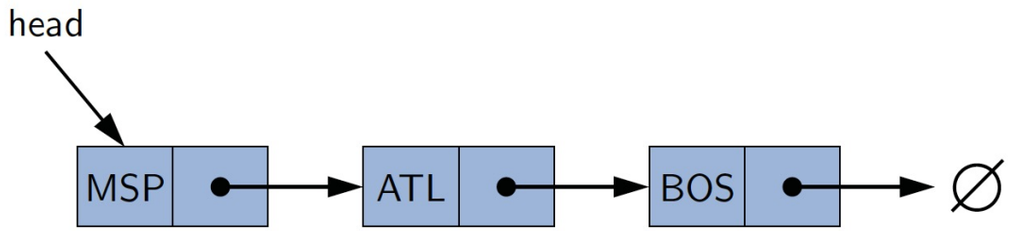
- Time complexity?



(a)



(b)



Removing i-th Element

- remove(i)
  - First arrive at the i-th node
  - Then update the links as you learned earlier
  - Time complexity?

Removing the Last Element

- removeLast()
- How will you implement this method?
- Time complexity?

Home Practice

- Some other operations to implement for practice
  - Addition between nodes
  - Removal between nodes
  - Addition after a certain value
  - Removal after a certain value
  - …

When List is Full

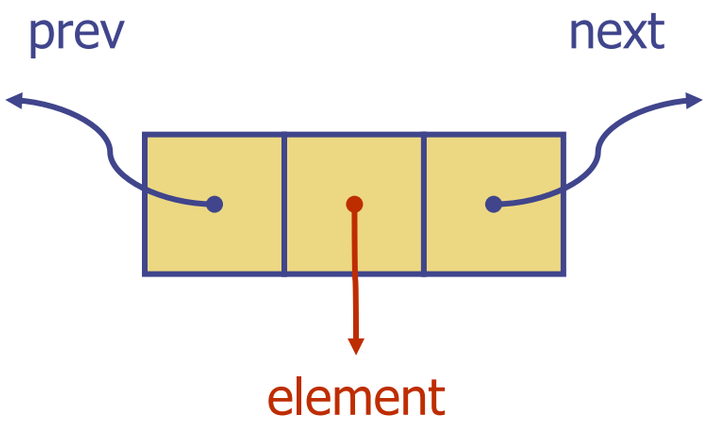
- add(i,e), or addLast(e)
  - What do you think will happen if a singly-linked list is full?

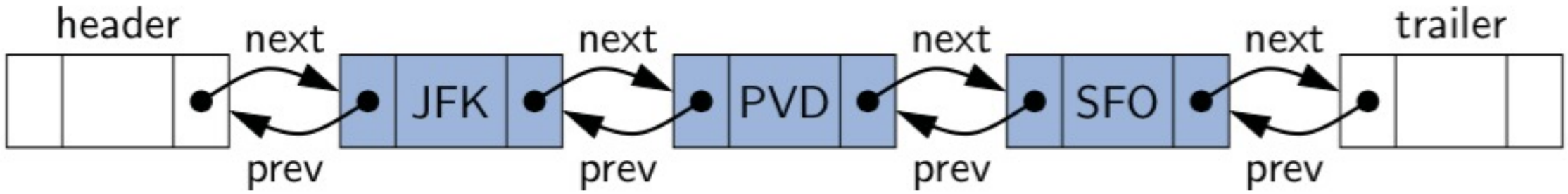
Analysis-Summary

Method	Running Time	
	Array-List	SLL
size()	O(1)	O(1)
isEmpty()	O(1)	O(1)
get(i)	O(1)	O(n)
set(i, e)	O(1)	O(n)
add(i, e)	O(n)	O(n)
remove(i)	O(n)	O(n)
addFirst(e)	O(n)	O(1)
addLast(e)	O(n)	O(n)
removeFirst()	O(n)	O(1)
removeLast()	?	?

Doubly Linked List

Doubly Linked List (DDL)





- Each node contains a value, a link to its successor (if any), *and* a link to its predecessor (if any)
- **The header** points to the first node in the list *and* **the trailer** to the last node in the list

List Operation in DLL

- Think about how these operations are performed in DLL and analyze their time complexity
  - set(i,e) and get(i)
  - Addition (at i-th position, at front, at the tail)
  - Removal (at i-th position, at front, at the tail)

Analysis-Summary

Method	Running Time		
	Array-List	SLL	DLL
size()	O(1)	O(1)	O(1)
isEmpty()	O(1)	O(1)	O(1)
get(i)	O(1)	O(n)	O(n)
set(i, e)	O(1)	O(n)	O(n)
add(i, e)	O(n)	O(n)	O(n)
remove(i)	O(n)	O(n)	O(n)
addFirst(e)	O(n)	O(1)	O(1)
addLast(e)	O(n)	O(n)	O(1)
removeFirst()	O(n)	O(1)	O(1)
removeLast()	?	?	?

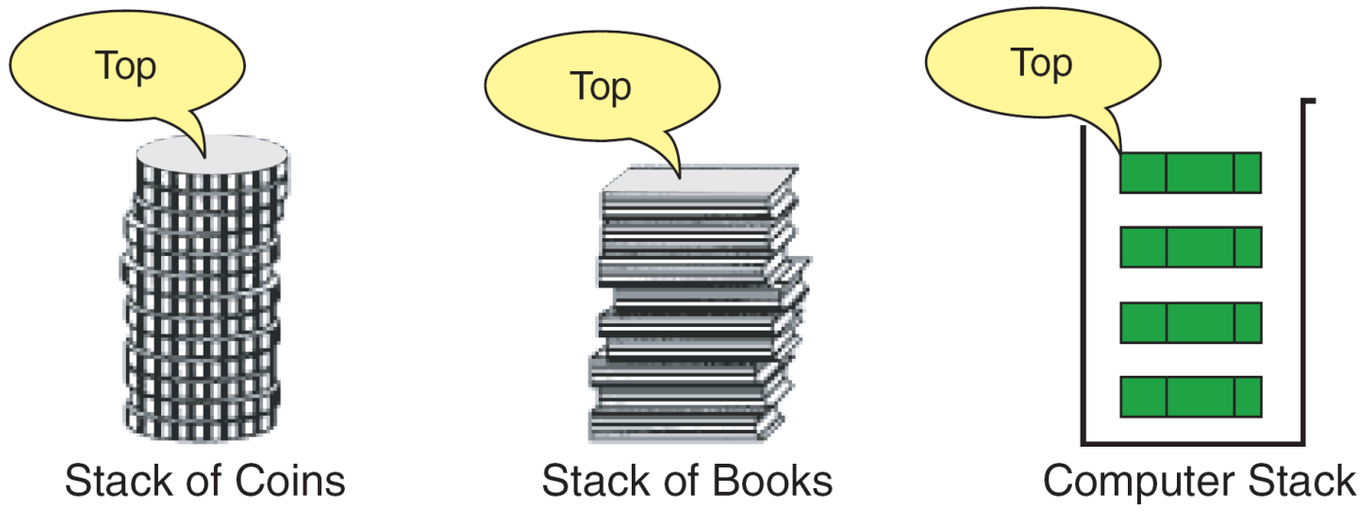
Stacks

Stacks

- A special kind of list
  - Addition and Removal takes place only at one end, called the top
  - the last element added, is always the first one to be deleted
- So, stack is a LIFO sequence (Last-In, First-Out)

Examples

- Mail in a pile on your desk
- Hyperlinks in your browser
- Method calls



Stack Operations

- size: returns the number of items in the stack
- isEmpty: returns whether stacks has no items
- top: returns the item at the top (without removing it)
- push: insert an item at the top of the stack
- pop: remove an item from the top of the stack

Stack Implementation

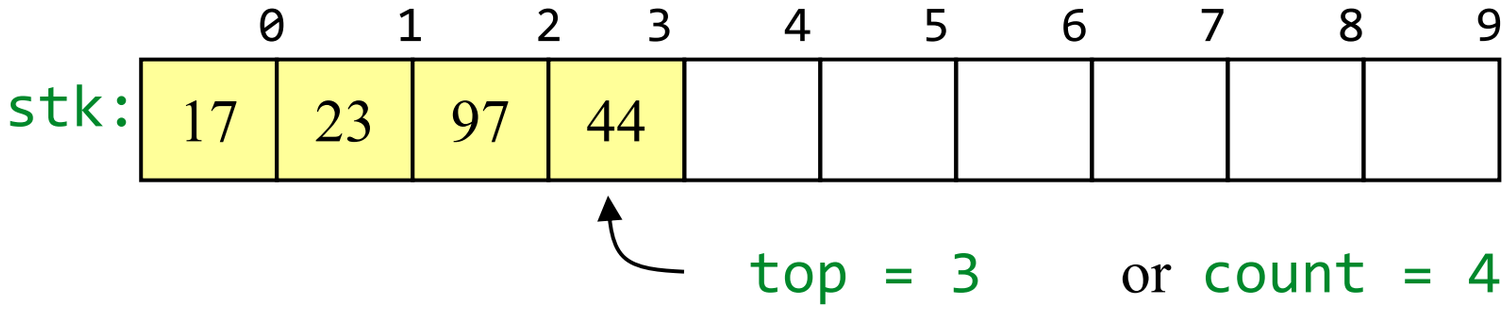
- Dedicated Implementation, OR
- All the operations can be directly implemented using the LIST ADT (as a wrapper around a built-in list object)

Stack ADT

```
public interface Stack<E> {
 int size();
 boolean isEmpty();
 E top();
 void push(E element);
 E pop();
}
```

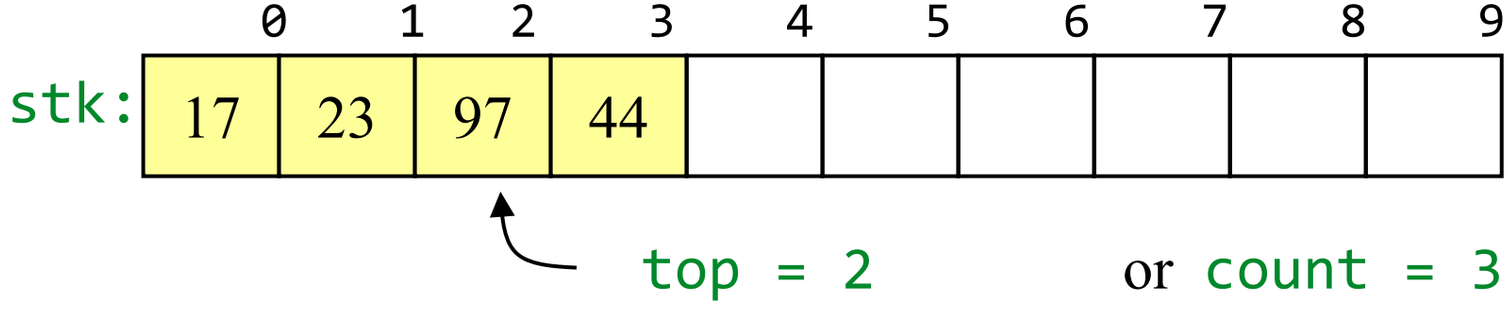
Array implementation of stacks

Pushing and popping



- If the bottom of the stack is at location `0`, then an empty stack is represented by `top = -1` or `count = 0`
- To add (push) an element, either:
  - Increment `top` and store the element in `stk[top]`, or
  - Store the element in `stk[count]` and increment `count`
- To remove (pop) an element, either:
  - Get the element from `stk[top]` and decrement `top`, or
  - Decrement `count` and get the element in `stk[count]`

After popping



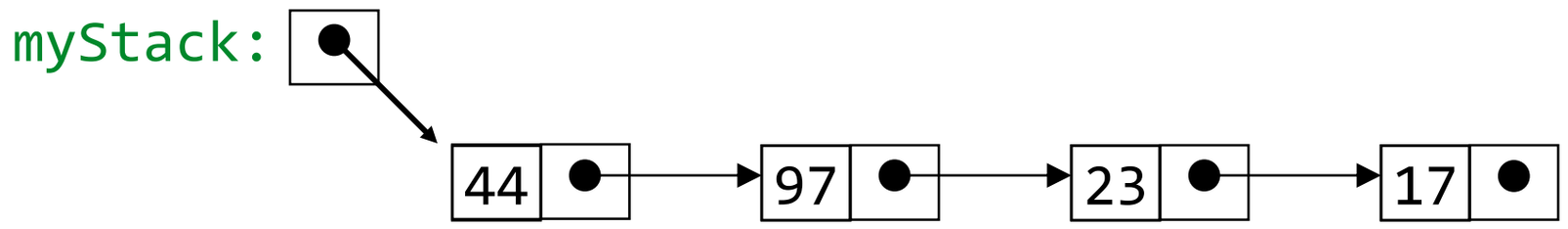
- When you pop an element, do you just leave the "deleted" element sitting in the array?
- The surprising answer is, *"it depends"*
  - If this is an array of primitives, *then* doing anything more is just a waste of time
  - If the array contains objects, you should set the "deleted" array element to `null`
  - Why? To allow it to be garbage collected!

ArrayStack

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
top	$O(1)$
push	$O(1)$
pop	$O(1)$

Linked-list implementation of stacks

- Since all the action happens at the top of a stack, a singly-linked list (SLL) is a fine way to implement it
- The header of the list points to the top of the stack



- Pushing is inserting an element at the front of the list
- Popping is removing an element from the front of the list

LinkedStack

Stack Method	*Singly Linked List Method*
size( )	list.size( )
isEmpty( )	list.isEmpty( )
push(e)	list.addFirst(e)
pop( )	list.removeFirst( )
top( )	list.first( )

Method	**Running Time**
size	$O(1)$
isEmpty	$O(1)$
top	$O(1)$
push	$O(1)$
pop	$O(1)$

Queues

Queues

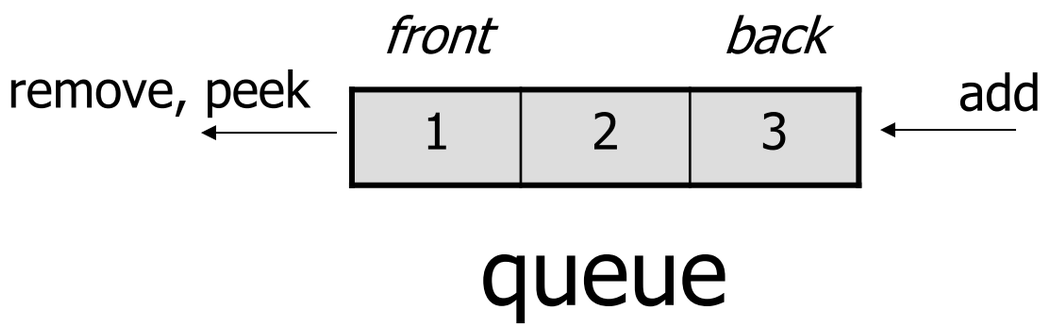
- A typical reason to use a queue in an application is to save the item in a collection while at the same time *preserving their relative order*

Queues

- Policy of doing tasks in the order they arrive
- Print jobs sent to the printer
- People waiting in line at a theater
- Cars waiting in line at a toll booth
- Tasks waiting to be serviced by an application on your computer

Queues

- Another special kind of list
  - Additions are made at one end, called the tail
  - Removals take place at the other end, called the head
  - the first element added, is always the first one to be deleted
- So, queue is a FIFO sequence



queue

Queue Operations

- size: returns the number of items in the stack
- isEmpty: returns whether stacks has no items
- first: returns the item at the head (without removing it)
- enqueue: insert an item at the *rear* of the queue
- eequeue: remove an item from the *front* of the queue

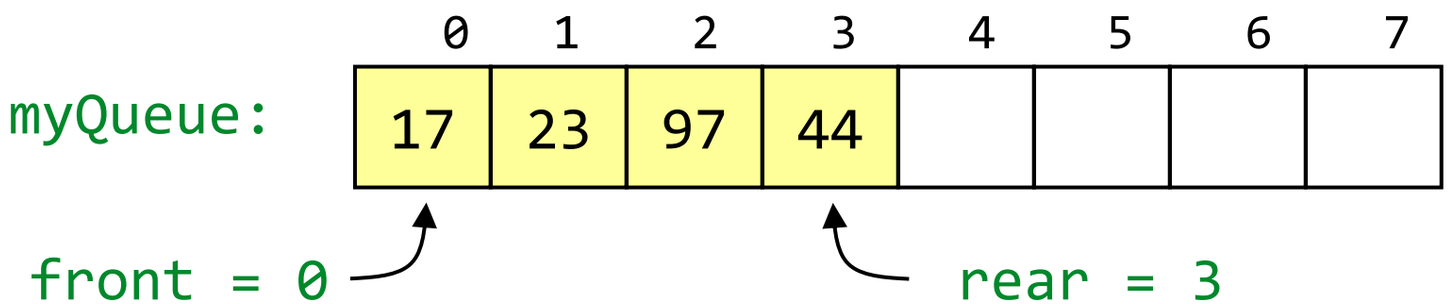
Queue Implementation

- Dedicated implementation, OR
- All the operations can be directly implemented using the LIST ADT (as a wrapper around a built-in list object)

Queue ADT

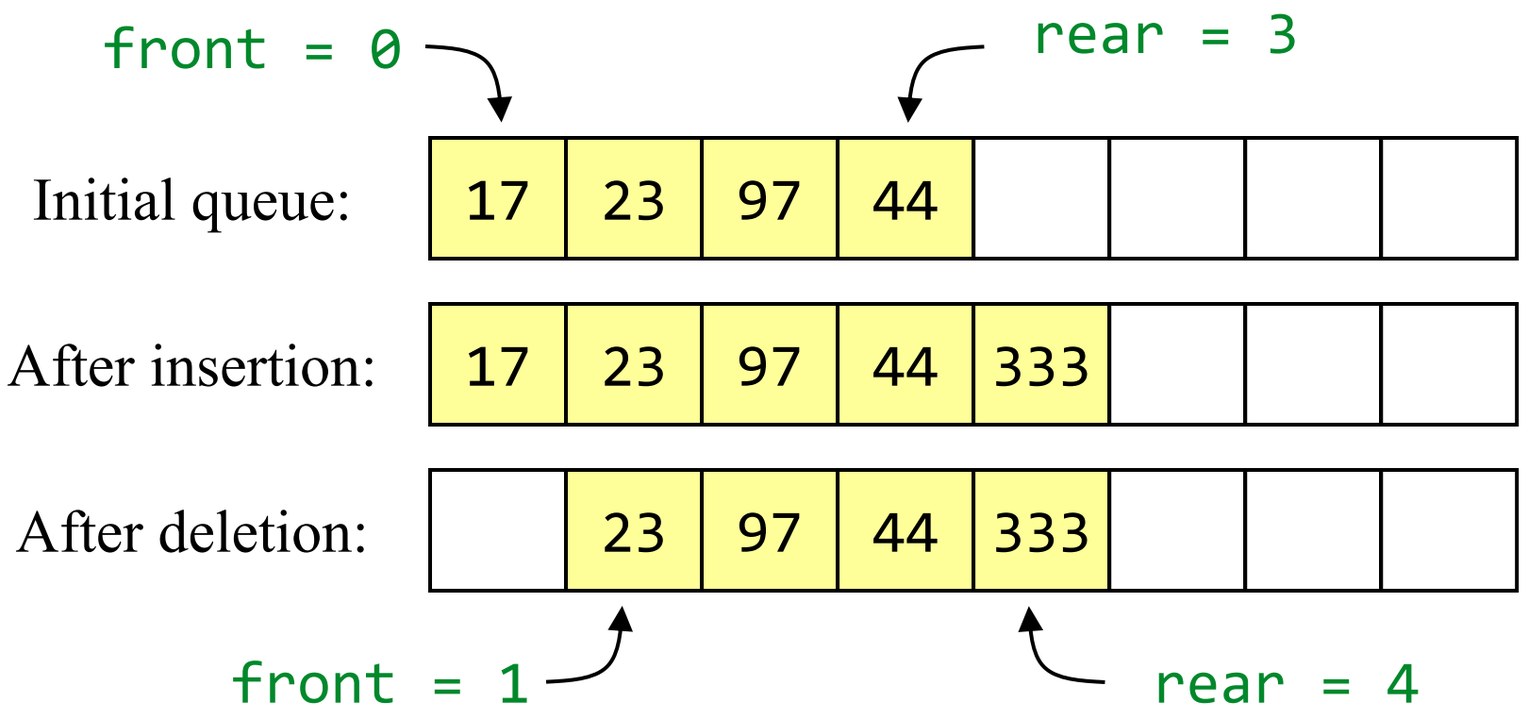
```
public interface Queue<E>
{
 int size();
 boolean isEmpty();
 E first();
 void enqueue(E e);
 E dequeue();
}
```

Array implementation of queues



- **To insert:** put new element in location 4, and set rear to 4
- **To delete:** take element from location 0, and set front to 1

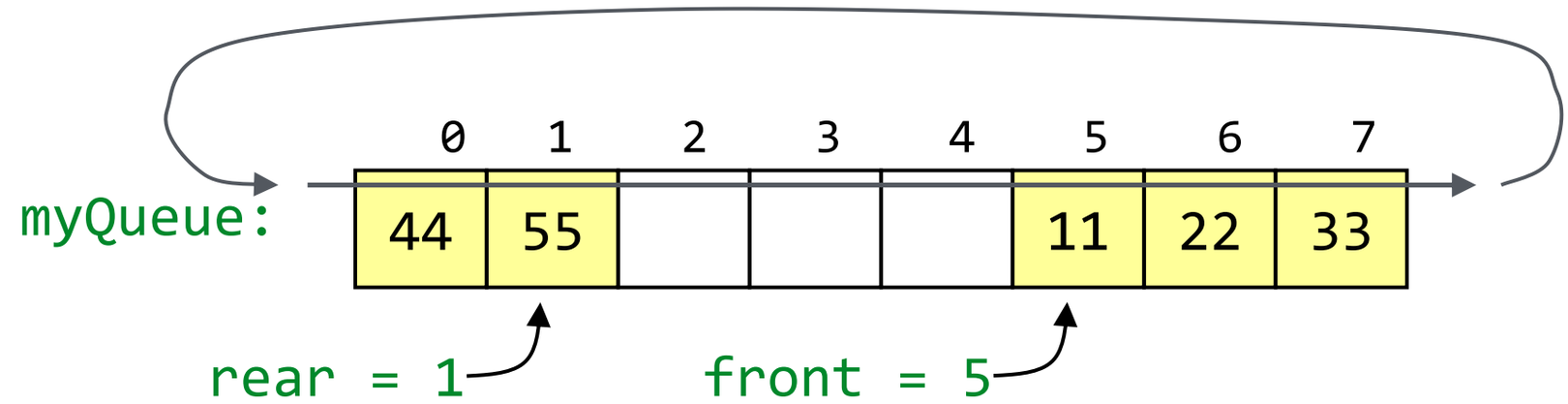
Array implementation of queues



- Notice how the array contents "crawl" to the right as elements are inserted and deleted
- This will be a problem after a while!

Circular arrays

- We can treat the array holding the queue elements as circular (joined at the ends)

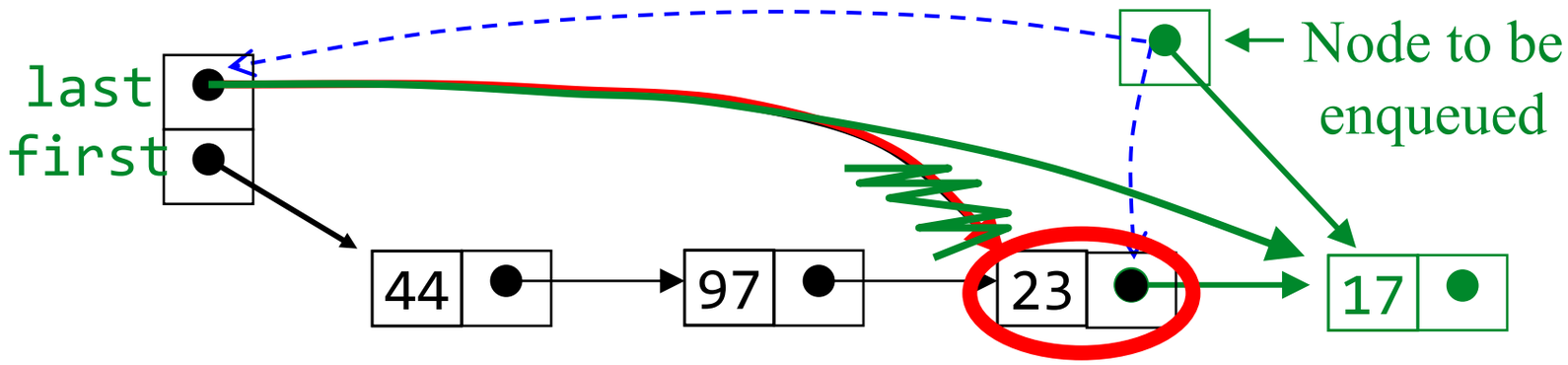


- Elements were added to this queue in the order `11`, `22`, `33`, `44`, `55`, and will be removed in the same order
- Use: `front = (front + 1) % myQueue.length;`
  and: `rear = (rear + 1) % myQueue.length;`

Linked-list implementation of queues

- In a queue, insertions occur at one end, deletions at the other end
- Operations at the front of a singly-linked list (SLL) are O(1), and at the other end they are
  - O(n) – if no special pointer to the last node
  - O(1) – If a pointer to the last node is kept

Enqueueing a node



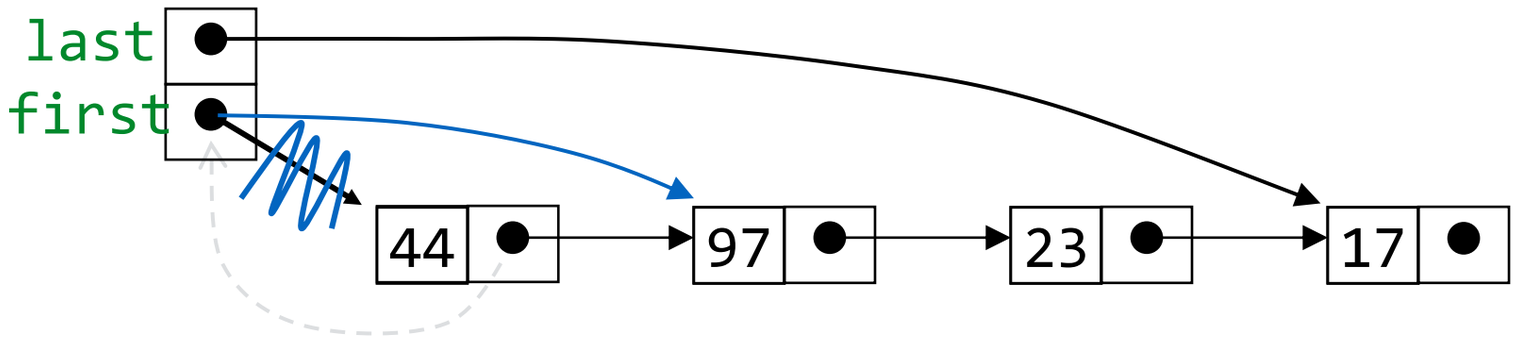
To enqueue (add) a node:

Find the current last node

Change it to point to the new last node

Change the `last` pointer in the list header

Dequeueing a node



- To dequeue (remove) a node:
  - Copy the pointer from the first node into the header

ArrayQueue

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
first	$O(1)$
enqueue	$O(1)$
dequeue	$O(1)$

LinkedQueue

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
first	$O(1)$
enqueue	$O(1)$
dequeue	$O(1)$

Did We Achieve Today's Objectives

- Abstract Data Type vs. Data Structure
- Basic building blocks of any data structure
- List ADT: its implementations and analysis
- Stack ADT: its implementations and analysis
- Queue ADT: its implementations and analysis