# CS121 Data Structures A, C
# Array Lists

Varduhi Yeghiazaryan
vyeghiazaryan@aua.am



Fall 2021

# Lists

We explore several abstract data types that represent

a linear sequence of elements, with more general support for adding or removing elements at arbitrary positions,

unlike the stack, queue and deque ADTs.

An index of an element $e$ in a sequence is equal to the number of elements before $e$ in that sequence.

# The List Abstract Data Type

The list ADT, based on `java.util.List`, supports:

size(): Returns the number of elements in the list

isEmpty(): Returns a boolean indicating whether the list is empty

get($i$): Returns the element of the list having index $i$; an error condition occurs if $i$ is not in range $[0, \text{size}() - 1]$

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 $[0, \text{size}() - 1]$

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 $[0, \text{size}()]$

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 $[0, \text{size}() - 1]$

# Example

| Method | Return Value | List Contents |
|:---:|:---:|:---:|
| add(0, A) | – | (A) |
| add(0, B) | – | (B, A) |
| get(1) | A | (B, A) |
| set(2, C) | "error" | (B, A) |
| add(2, C) | – | (B, A, C) |
| add(4, D) | "error" | (B, A, C) |
| remove(1) | A | (B, C) |
| add(1, D) | – | (B, D, C) |
| add(1, E) | – | (B, E, D, C) |
| get(4) | "error" | (B, E, D, C) |
| add(4, F) | – | (B, E, D, C, F) |
| set(2, G) | D | (B, E, G, C, F) |
| get(2) | G | (B, E, G, C, F) |

# The List Application Programming Interface (API)

```
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  }
```

# Array Lists

A simple way of implementing the list ADT uses an array $A$, where $A[i]$ stores (a reference to) the element with index $i$

Using an advanced technique allows to have unbounded lists known as **array lists** in Java or **vectors** in C++

With a representation based on an array $A$, the $\texttt{get}(i)$ and $\texttt{set}(i, e)$ methods are easy to implement by accessing $A[i]$ (assuming $i$ is a legitimate index)
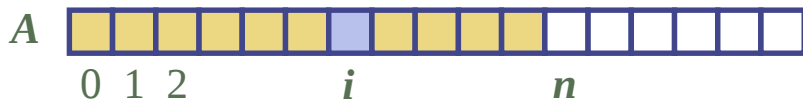
$A$

0 1 2      $i$      $n$

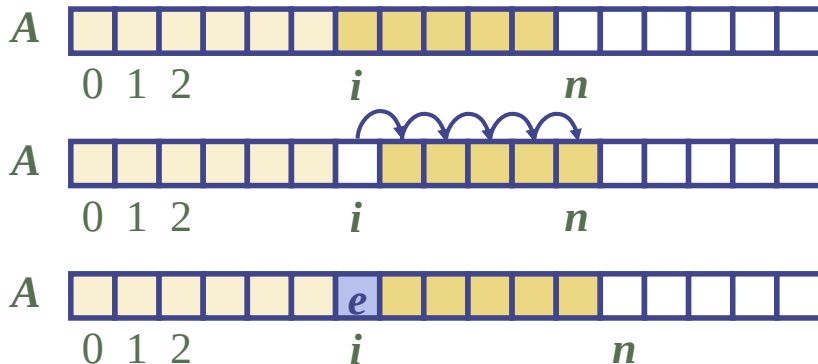# Array Lists: Element Insertion

add($i, e$) needs to make room for the new element by shifting forward the $n - i$ elements $A[i], \ldots, A[n-1]$
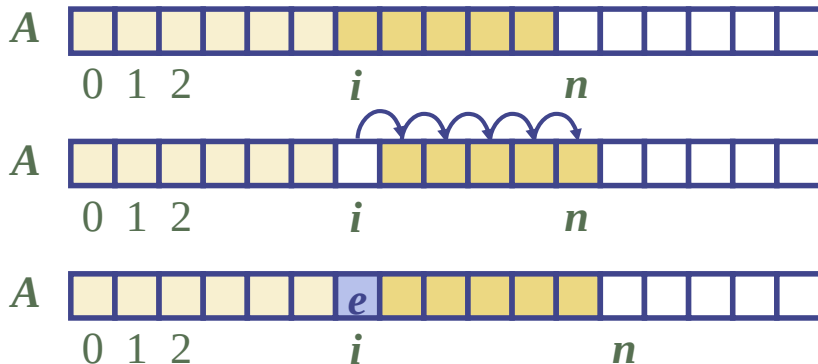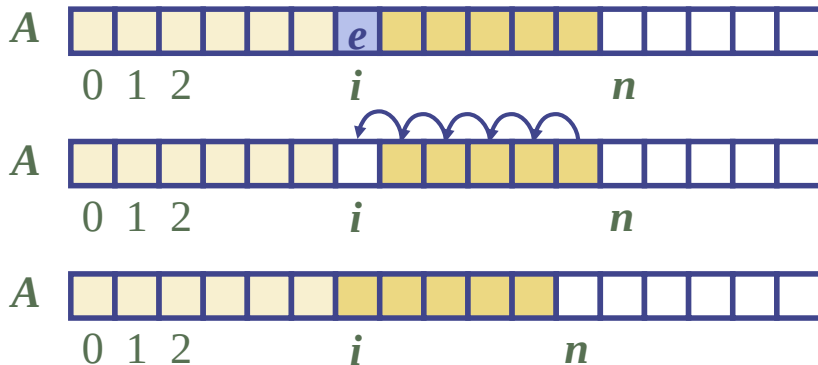
# Array Lists: Element Insertion

add($i, e$) needs to make room for the new element by shifting
forward the $n - i$ elements $A[i], \ldots, A[n-1]$



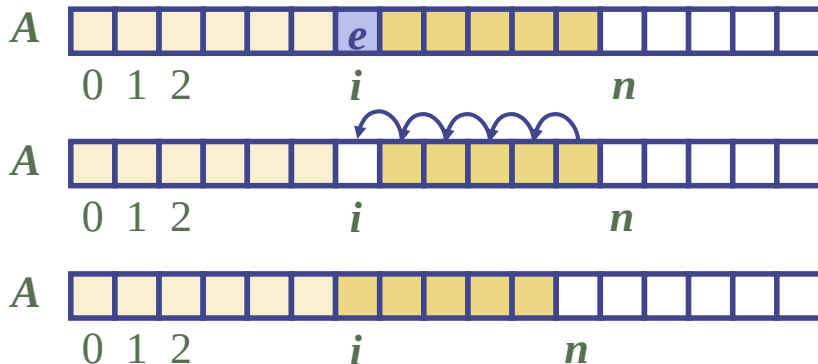In the worst case ($i = 0$), element insertion takes $O(n)$ time

# Array Lists: Element Removal

remove($i$) needs to fill the hole left by the removed element by shifting backward the $n - i - 1$ elements $A[i + 1], \ldots, A[n - 1]$

# Array Lists: Element Removal

remove($i$) needs to fill the hole left by the removed element by shifting backward the $n - i - 1$ elements $A[i + 1], \ldots, A[n - 1]$



In the worst case ($i = 0$), element removal takes $O(n)$ time

# Array List Simple Implementation I

```
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    }
11    // public methods
12    /** Returns the number of elements in the array list. */
13    public int size( ) { return size; }
14    /** Returns whether the array list is empty. */
15    public boolean isEmpty( ) { return size == 0; }
16    /** Returns (but does not remove) the element at index i. */
17    public E get(int i) throws IndexOutOfBoundsException {
18      checkIndex(i, size);
19      return data[i];
20    }
```

# Array List Simple Implementation II

```
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    }
28    /** Inserts element e to be at index i, shifting all subsequent elements later. */
29    public void add(int i, E e) throws IndexOutOfBoundsException, IllegalStateException {
30      checkIndex(i, size + 1);
31      if (size == data.length)                    // not enough capacity
32        throw new IllegalStateException("Array is full");
33      for (int k=size−1; k >= i; k−−)             // start by shifting rightmost
34        data[k+1] = data[k];
35      data[i] = e;                                // ready to place the new element
36      size++;
37    }
```

# Array List Simple Implementation III

```
38    /** Removes/returns the element at index i, shifting subsequent elements earlier. */
39    public E remove(int i) throws IndexOutOfBoundsException {
40      checkIndex(i, size);
41      E temp = data[i];
42      for (int k=i; k < size−1; k++)              // shift elements to fill hole
43        data[k] = data[k+1];
44      data[size−1] = null;                        // help garbage collection
45      size−−;
46      return temp;
47    }
48    // utility method
49    /** Checks whether the given index is in the range [0, n−1]. */
50    protected void checkIndex(int i, int n) throws IndexOutOfBoundsException {
51      if (i < 0 || i >= n)
52        throw new IndexOutOfBoundsException("Illegal index: " + i);
53    }
54  }
```

# Array List Simple Implementation: Analysis

*Drawback:* fixed-capacity array, limiting the ultimate list size

If the application needs much less space than the reserved capacity, memory is wasted

If we try to add an element into a full array, the implementation throws an exception and refuses to store the new element

| Method | Time |
|---:|:---:|
| size() | $O(1)$ |
| isEmpty() | $O(1)$ |
| get($i$) | $O(1)$ |
| set($i, e$) | $O(1)$ |
| add($i, e$) | $O(n)$ |
| remove($i$) | $O(n)$ |

Space usage: $O(N)$, where $N$ is the size of the array, independent from the number $n \leq N$ of elements in the list
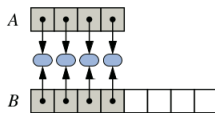
# Dynamic Arrays

*Why can't we increase the capacity of an array after its allocation?*

*Idea:* When a user tries to add into a full list, i.e. all reserved capacity in the underlying array is exhausted, replace the underlying array with a larger one.
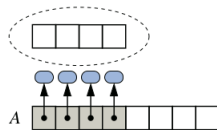
1. Allocate a new array $B$ with larger capacity
2. Set $B[k] = A[k]$, for $k = 0, \ldots, n-1$, where $n$ denotes current number of items
3. Set $A = B$, that is, we henceforth use the new array to support the list
4. Insert the new element in the new array
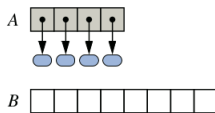


(a)          (b)          (c)

# Array List Implementation with a Dynamic Array

Include the `resize` method below as a protected method within the original `ArrayList` class.

```
/** Resizes internal array to have given capacity >= size. */
protected void resize(int capacity) {
  E[ ] temp = (E[ ]) new Object[capacity];      // safe cast; compiler may give warning
  for (int k=0; k < size; k++)
    temp[k] = data[k];
  data = temp;                                   // start using the new array
}
```

Redesign the `add` method so that it calls the new `resize` utility when detecting that the current array is filled (rather than throwing an exception)

```
28  /** Inserts element e to be at index i, shifting all subsequent elements later. */
29  public void add(int i, E e) throws IndexOutOfBoundsException {
30    checkIndex(i, size + 1);
31    if (size == data.length)              // not enough capacity
32      resize(2 * data.length);            // so double the current capacity
33    // rest of method unchanged...
```

# Analysis of Strategies for Dynamic Arrays

Let the operation of adding an element at the end of the list be called **push**, as a shorthand notation

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

We compare the incremental strategy and the doubling strategy by analysing 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 a 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

$$n + c + 2c + 3c + 4c + \cdots + kc =$$
$$n + c(1 + 2 + 3 + \cdots + k) =$$
$$n + ck(k + 1)/2$$

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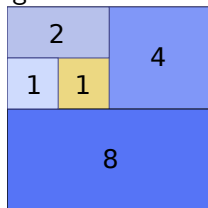
Thus, the amortized time of a push operation is $O(n)$

# 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

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

geometric series



$T(n)$ is $O(n)$

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

# Summary

**Reading**

Section 7.1 The List ADT

Section 7.2 Array Lists

**Questions?**