CS121 Data Structures
Array Lists

Monika Stepanyan
mstepanyan@aua.am

Spring 2024

Important Data and Statistics

- $\frac{1}{3}$ of the semester passed!
- 20 classes remaining
- 18 days till the Midterm exam I
- 9 days till the Spring Break
- HW1 due Sunday, February 25, 23:59

Important Data and Statistics: Midterm I

The topics included in the exam– slides 01–07 (intro–deque)

The question types breakdown:

- 50% theory
- 50% programming

To Prepare for the Midterm I

- You should read chapters 1, 2, 3, 4, 5, 6, 12 or do any equivalent activity for enhancing your theoretical understanding
- You should do coding and OOP revision (chapters 1 and 2)
- You should practice recursion tracing (including Sorting Trees)
- You should be able to simulate different sorting algorithms on a sequence of elements.
- You should know the running times of sorting algorithms (worst case, best case, average case) and the corresponding inputs for those.

To Prepare for the Midterm I

- You should know
  - and be able to implement all the ADTs and their concrete implementations.
  - all the public methods and instance variables of the ADTs and their concrete implementations.
  - time and space complexities for all methods of ADTs and their concrete implementations.
- You should be able to implement alternative concrete versions of ADTs.
- You should be able to analyze the time and space complexity for those.
- You should be able to solve problems with concrete data structures.

Lists

List is an abstract data type that represents a *linear* sequence of elements, with more *general* support for adding or removing elements at arbitrary positions (unlike the stack, queue and deque ADTs).

Each element in a list is associated with a unique index.

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)

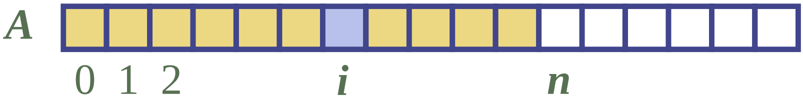
```
/** A simplified version of the java.util.List interface. */
public interface List<E> {
  /** Returns the number of elements in this list. */
  int size( );

  /** Returns whether the list is empty. */
  boolean isEmpty( );

  /** Returns (but does not remove) the element at index i. */
  E get(int i) throws IndexOutOfBoundsException;

  /** Replaces the element at index i with e, and returns the replaced element. */
  E set(int i, E e) throws IndexOutOfBoundsException;

  /** Inserts element e to be at index i, shifting all subsequent elements later. */
  void add(int i, E e) throws IndexOutOfBoundsException;

  /** Removes/returns the element at index i, shifting subsequent elements earlier. */
  E remove(int i) throws IndexOutOfBoundsException;
}
```

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

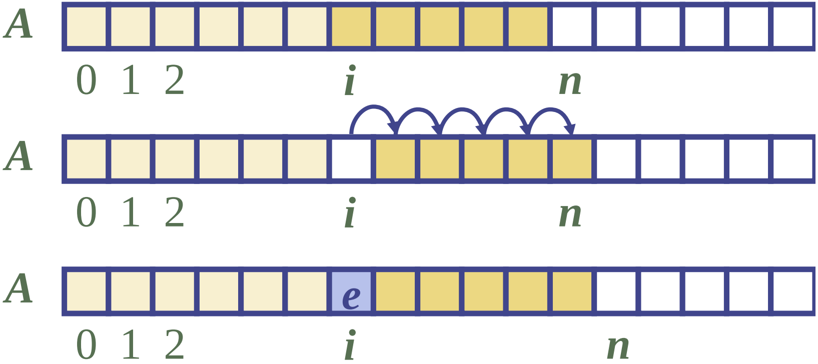
Using dynamic resizing allows to have unbounded lists.

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)



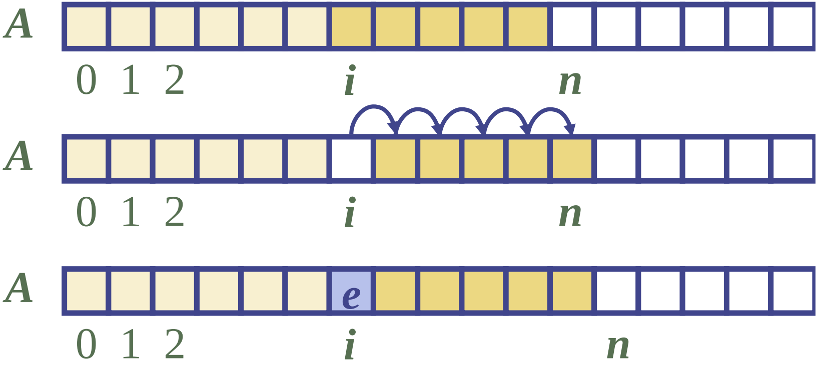
Array Lists: Element Insertion

`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

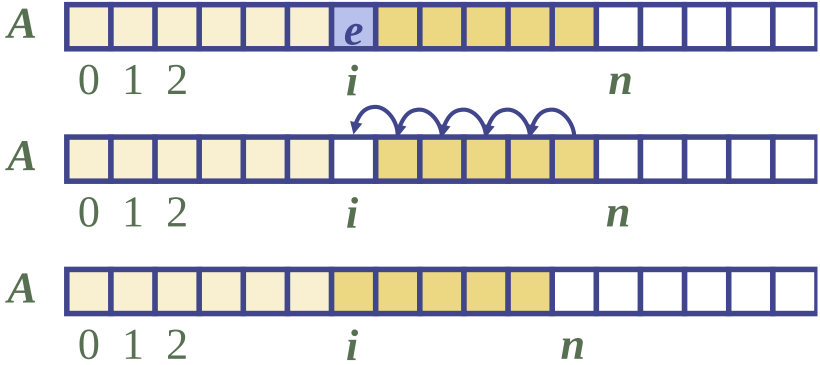
add(i, e) needs to make room for the new element by shifting forward the $n - i$ elements $A[i], \ldots, A[n-1]$



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

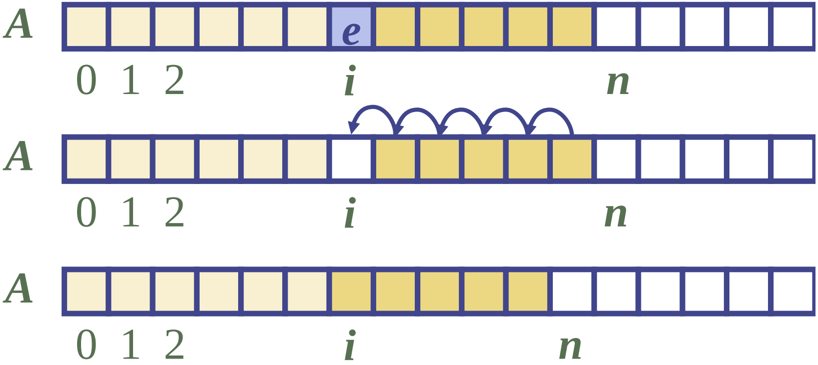
Array Lists: Element Removal

`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

```
public class ArrayList<E> implements List<E> {
  // instance variables
  public static final int CAPACITY = 16;        // default array capacity
  private E[ ] data;                            // generic array used for storage
  private int size = 0;                         // current number of elements
  // constructors
  public ArrayList( ) { this(CAPACITY); }       // constructs list with default capacity
  public ArrayList(int capacity) {              // constructs list with given capacity
    data = (E[ ]) new Object[capacity];         // safe cast; compiler may give warning
  }
  // public methods
  /** Returns the number of elements in the array list. */
  public int size( ) { return size; }
  /** Returns whether the array list is empty. */
  public boolean isEmpty( ) { return size == 0; }
  /** Returns (but does not remove) the element at index i. */
  public E get(int i) throws IndexOutOfBoundsException {
    checkIndex(i, size);
    return data[i];
  }
```

Array List Simple Implementation II

```
    /** Replaces the element at index i with e, and returns the replaced element. */
    public E set(int i, E e) throws IndexOutOfBoundsException {
      checkIndex(i, size);
      E temp = data[i];
      data[i] = e;
      return temp;
    }
    /** Inserts element e to be at index i, shifting all subsequent elements later. */
    public void add(int i, E e) throws IndexOutOfBoundsException, IllegalStateException {
      checkIndex(i, size + 1);
      if (size == data.length)                  // not enough capacity
        throw new IllegalStateException("Array is full");
      for (int k=size-1; k >= i; k--)           // start by shifting rightmost
        data[k+1] = data[k];
      data[i] = e;                              // ready to place the new element
      size++;
    }

  /** Inserts element e at the end of the list. */
  public void add(E e) throws IllegalStateException {
      add(size, e);
  }
```

Array List Simple Implementation III

```
  /** Removes/returns the element at index i, shifting subsequent elements earlier. */
  public E remove(int i) throws IndexOutOfBoundsException {
    checkIndex(i, size);
    E temp = data[i];
    for (int k=i; k < size-1; k++)              // shift elements to fill hole
      data[k] = data[k+1];
    data[size-1] = null;                        // help garbage collection
    size--;
    return temp;
  }
  // utility method
  /** Checks whether the given index is in the range [0, n-1]. */
  protected void checkIndex(int i, int n) throws IndexOutOfBoundsException {
    if (i < 0 || i >= n)
      throw new IndexOutOfBoundsException("Illegal index: " + i);
  }
}
```

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

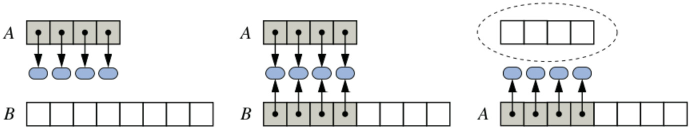
Method	**Time**
size()	$O(1)$
isEmpty()	$O(1)$
get(i)	$O(1)$
set(i, e)	$O(1)$
add(i, e)	$O(n)$
remove(i)	$O(n)$

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

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)

```
/** Inserts element e to be at index i, shifting all subsequent elements later. */
public void add(int i, E e) throws IndexOutOfBoundsException {
  checkIndex(i, size + 1);
  if (size == data.length)                      // not enough capacity
    resize(2 * data.length);                    // so double the current capacity
  // 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 a list represented by a growable array of some constant size (usually 0 or 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

We start with a full list of size c (the increment value)

Over n push operations, we replace the array $k = \lceil n/c \rceil$ times

The total time $T(n)$ of a series of n push operations is proportional to

$$
\begin{aligned}
& n + c + 2c + 3c + 4c + \cdots + kc = \\
& n + c(1 + 2 + 3 + \cdots + k) = \\
& n + c\frac{k(k+1)}{2}
\end{aligned}
$$

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

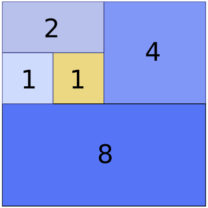
We start with a full list of size 1

We replace the array $k = \lceil \log_2(n+1) \rceil$ times

The total time $T(n)$ of a series of n push operations is proportional to

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

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?