CSE 12 – Basic Data Structures

Prof. Christine Alvarado
[Slides borrowed/adapted from slides by Cynthia Lee, Rakesh Varna, & Roshni Chandrashekhar]
Announcements

1. HW3 posted... have you started?
   A. No, not at all
   B. I’ve read it
   C. I’ve started solving the problems
   D. I’m (almost) done
Announcements

1. HW4 coming… start early!
2. Midterm conflicts: we’ll be nailing down an alternate time this week
2. When using push() to place the following items on a stack:
push(32)
push(65)
push(0)
push(23)
push(-1)
the output when popping from the stack is:

A. 32, 0, 23, -1, 65
B. 0, 23, -1, 32, 65
C. 32, 65, 0, 23, -1
D. -1, 23, 0, 65, 32
Stacks
Stacks

- A data structure that provides Last-In-First-Out semantics.

- Restricted operations to achieve LIFO principle.
Stacks

- Operations:
  - push – push element on top of the stack.
  - pop – remove element at the top of the stack
  - peek – return element at the top of the stack, without removing it.

- All operations are made on ‘top’ of the stack.
Stacks - Implementation

- Implement the methods in the ADT from scratch.
- Be clever about it.
Adapter Pattern
Lazy Christine needs to implement the Stack Interface with the following methods:

- push(E element) – add elements on to the stack.
- E pop() – remove element from top of the stack.
- E peek() – return the element at the top of the stack.

True to her name, she is looking for ways to avoid doing a lot of work.
Adapter Pattern

Christine comes to know that Phil has written a very good implementation of a Double-ended List interface – DList12 which provides the following methods:

- `addFront(E element)`
- `E removeFront()`
- `E peekFront()`
- `addBack(E element)`
- `E removeBack()`
- `E peekBack()`
Inheritance?

Christine realizes that she can just make Stack extend the List written by Phil and write the additional methods by using other existing methods.

Ex:

```java
public Stack12<E> extends DList12<E>
{
    ...
    public E pop() {
        return removeFront();  // A call to the removeFront
                                // inherited from DList12
    }
    ...
}
```

Pros? Cons? Discuss with your group
Inheritance?

Christine realizes that she can just make Stack extend the List written by Phil and write the additional methods by using other existing methods.

```java
public Stack12<E> extends DList12<E>
```

Which of the following is the biggest drawback of this approach:

A. It is inefficient from a running-time perspective: A double-ended linked list is not a good choice for a Stack implementation
B. It is incorrect. There is no way to implement all the methods in the Stack interface with the methods in the double linked list
C. It exposes methods that are not supposed to be part of the Stack interface.
Inheritance is not always the right answer

- Christine comes to know that Phil has written a very good implementation of a Double-ended List interface – DList12 which provides the following methods:
  - `addFront(E element)`
  - `E removeFront()`
  - `E peekFront()`
  - `addBack(E element)`
  - `E removeBack()`
  - `E peekBack()`
Inheritance

- Such an implementation comes with strings attached.
- The other methods in the List are public and accessible by anyone. But a Stack does not expose such methods! (Ex: addBack, removeBack() etc)
- So Inheritance is not the best design pattern to use here.
Adapter Pattern

I can use a private list variable

and use its methods within stack
Adapter Pattern

Making the List variable private makes sure that users of the Stack cannot access the List or its methods.

Only the Stack methods are public and therefore usable by clients.

You can happily use List within Stack and pass on operations to it.
public class Stack12<E> implements BoundedStack<E> {
    private Deque<E> stack;
    ...
    public E pop() {
        return stack.removeBack();
    }
}
And no one needs to know..

Every one thinks I implemented Stack from scratch
Mapping Attributes

- Before deciding on what methods to use, one needs to map the corresponding attributes.

- For example: To use the List as a Stack, we need to map the Top of the stack to some position in the list (front or back—our choice, but how to choose?)
Mapping methods

- Once this is done, we can map the methods on top of the stack to methods operating on the head of the List.

  If we choose the front....
  - push -> addFront
  - pop -> removeFront
  - peek -> peekFront
Adapter Pattern Summary

You would like to implement an Interface A.

You have an implementation B that implements another interface C which defines methods very much similar to the methods in A but differ slightly (like name).

You use an instance of B inside your class that implements A and delegate tasks to it.
Stack

Using the Adapter pattern with different lists
Mapping between ArrayList and Stack (Stack “has a” ArrayList)

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

What is the time cost of adding or removing an element at the head or at the tail of an N-element List...

- If List is implemented using an array?
  - Head (index 0): _____  Tail (index size-1): _____

A. $O(1)$, $O(1)$
B. $O(1)$, $O(n)$
C. $O(n)$, $O(n)$
D. $O(n^2)$, $O(n^2)$
E. Other/none/more
Mapping between single linked list and Stack (Stack “has a” single linked list)

What is the time cost of adding or removing an element at the head or at the tail of an N-element List...

- If List is implemented using a singly linked list?
  - Head: _____  Tail (no direct pointer): _____

A.  O(1), O(1)
B.  O(1), O(n)
C.  O(n), O(n)
D.  O(n^2), O(n^2)
E.  Other/none/more
What is the time cost of adding an element at the head or at the tail of an N-element List...

- If List is implemented using a singly linked list?
  Head: _____  Tail (with direct pointer): _____

A. $O(1), O(1)$
B. $O(1), O(n)$
C. $O(n), O(n)$
D. $O(n^2), O(n^2)$
E. Other/none/more
What is the time cost of removing an element at the head or at the tail of an N-element List...

If List is implemented using a singly linked list?

Head: _____ Tail (with direct pointer): _____

A. \( O(1), O(1) \)
B. \( O(1), O(n) \)
C. \( O(n), O(n) \)
D. \( O(n^2), O(n^2) \)
E. Other/none/more
What is the time cost of adding or removing an element at the head or at the tail of an N-element List...

If List is implemented using a doubly linked list?
Head: _____ Tail (with direct pointer): ____

A. O(1), O(1)
B. O(1), O(n)
C. O(n), O(n)
D. O(n^2), O(n^2)
E. Other/none/more
Map Stack Attributes to ArrayList Attributes and/or Methods

<table>
<thead>
<tr>
<th>Stack Attribute</th>
<th>ArrayList Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>size() – 1</td>
</tr>
<tr>
<td>size</td>
<td>size()</td>
</tr>
</tbody>
</table>

Don’t underestimate the importance of doing this mapping first. Planning now saves time later.
A consequence of that attribute mapping is that a push operation results in adding to the tail of the List.

\[
\text{size()} - 1 + 1 = \text{size()}
\]

<table>
<thead>
<tr>
<th>Stack operation</th>
<th>List operation equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>push( element )</td>
<td>add( size(), element )</td>
</tr>
<tr>
<td>E pop()</td>
<td>E remove( size() - 1 )</td>
</tr>
<tr>
<td>E peek()</td>
<td>E get( size() - 1 )</td>
</tr>
<tr>
<td>int size()</td>
<td>int size()</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>boolean isEmpty()</td>
</tr>
</tbody>
</table>
Stack Implementations, done correctly...

<table>
<thead>
<tr>
<th>Operation</th>
<th>LinkedList (head = TOS)</th>
<th>ArrayList (end = TOS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push(E element)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Pop()</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Peek()</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>