Lecture 9

Announcements

- Rest is over: HW4 is out. It is LONG and has a lot of pieces that you need to put together.
  - 2 milestone submissions to get you started.
  - Deadline is a bit longer than a week. (till next Tuesday)

- Then you will have HW5 (about sorts), not a long one again.
  - + midterm.

Today's Plan

- The Stack ADT
- Building a Stack using the Adapter Pattern Design
- How underlying data structure (and choices of mapping) affects complexity.

Stacks and Queues

- Stacks and queues are examples of *linear* data structures in which every object inserted into it will generally be removed.
- Usually a stack/queue is intended only as “temporary” storage.
- Both stacks and queues allow the user to **add** and **remove** elements.
- Not good for sorting.
- Where they differ is the *order* in which elements are removed *relative to when they were added.*

Stack of crepes

Stacks (LIFO)

- Stacks are *last-in-first-out* (**LIFO**) data structures.

- The classic analogy for a "stack" is a pile of dishes:
  - Suppose you've already added dishes A, B, and C to the "stack" of dishes.
  - Now you add one more, D.
  - Now you remove one dish -- *you get D back.*
  - If you remove another, you get C, and so on.
- With stacks, you can only add to/remove from the *top* of the stack.

Warm Up

- When using push() operation to place the following items on a stack:

push(10)

push(20)

push(30)

push(0)

push(-30)

the output when popping from the stack is:

A: 10, 20, 30, 0 , -30

B: -30, 0, 10, 20, 30

C: 30, 10, 20, 0, -30

D: -30, 0, 30, 20, 10

E: 0, 30, -30, 10, 20

Stacks

- **Operations**:
  - Push – push (add) element on top of the stack
  - Pop – pop (remove) element from the top of the stack
  - Peek – return element at the top of the stack, without removing it
  - *What else could be useful?*

- All operations are made on **top** of the stack

Stack ADT

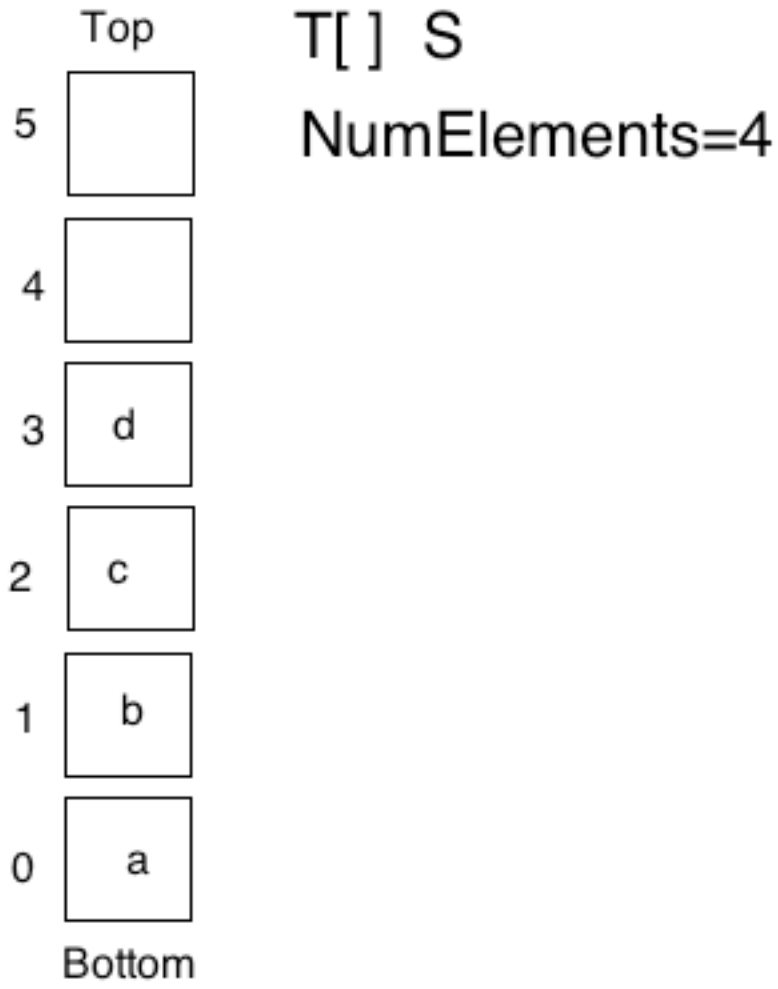
- To support the **LIFO** adding/removal of elements, a stack must adhere to the following interface:

```
interface Stack<E> {
  //Adds the specified object to the top of the stack
  void push (E o);

  //Removes the top of the stack and returns it
  E pop ();

  //Returns the top of the stacj without removing it
  E peek ();
}
```

Stack ADT

- Stack can be implemented using **two** kinds of storage:
  - **Array**: More efficient for stacks of a fixed maximum capacity.
  - **Linked list**: More flexible for stacks with a "growable" capacity.

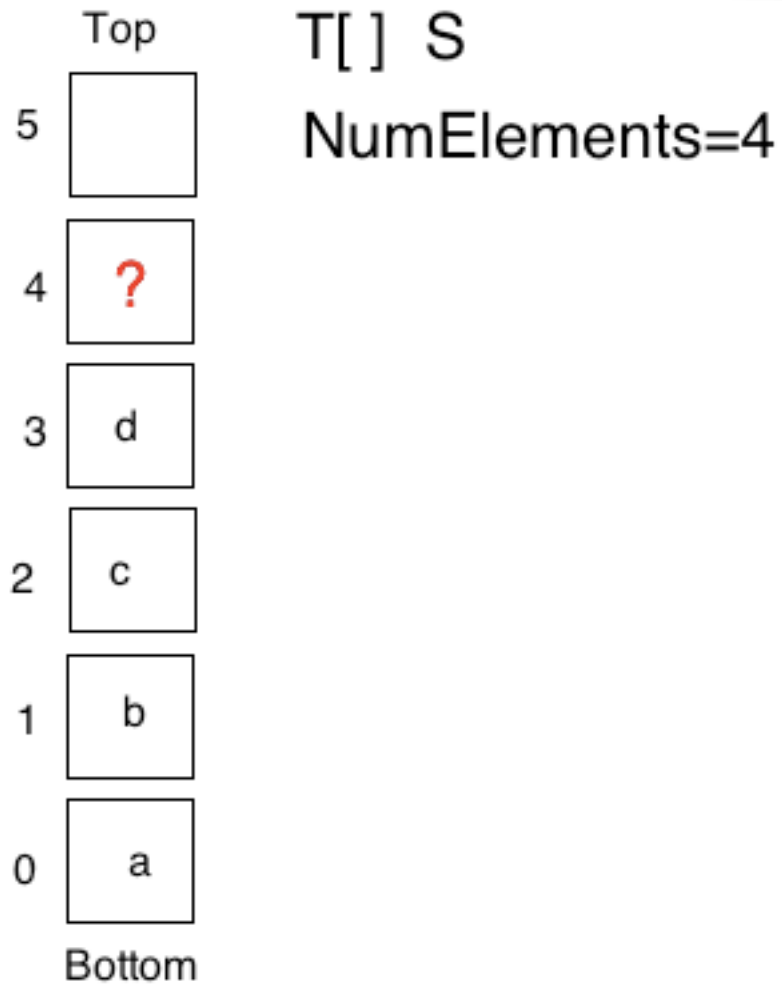
Array-based stacks



Array-based stacks

- **Push** ?

- A: S[S.length]=e;
- B: S[NumElements]=e;
- C: S[NumElements+1]=e;
- D: S[NumElements-1]=e;
- E: Other



Exceptions

- If you are to implement an array-based stack, how many exceptions would you throw?

- A: None
- B: 1
- C: 2
- D: 3
- E: More than 3

Exceptions

- If a stack has reached its maximum capacity and the user calls **push(o),** then the stack will **overflow.**

- If a stack is empty (numElements **== 0)** and the user calls **pop(),** then the stack will **underflow.**

Stack Complexity

- What is complexity of the array-based stack operations?
  - *Assume there is room to add and there are elements to remove*.

- *Push*, then *Pop*

- A: O(1)
- B: O(n)
- C: O(log n)
- D: $O(n^2)$
- E: Other

STACK AS A LINKED LIST

Linked list-based stacks

```
public class myStack<E> {
    protected class StackNode {
        E element;
        StackNode next;
    }

    private StackNode top;
    public boolean push( E elem ) {
        StackNode n = new StackNode();
        n.next = top;
        n.element = elem;
        top = n;
        return true;
    }
}
```

One option: Implement the methods in the ADT from scratch.

```
public class myStack<E> {
    protected class StackNode {
        E element;
        StackNode next;
    }

    private StackNode top;
    public boolean push( E elem ) {
        StackNode n = new StackNode();
        n.next = top;
        n.element = elem;
        top = n;
        return true;
    }
}
```

```
public class myList<E> {
    protected class ListNode {
        E element;
        ListNode next;
    }

    private ListNode head;
    public boolean addFirst( E elem ) {
        ListNode n = new ListNode();
        n.next = head;
        n.element = elem;
        head = n;
        return true;
    }
}
```

Why redo all this work??

ADAPTER PATTERN DESIGN

Adapter Pattern

- We need to implement the **Stack** interface with the following methods:
  - *push* (E element) - add element on the stack
  - *E pop()* – remove element from top of the stack
  - *E peek()* – return the element at the top of the stack

- We would like to find a way to reduce our work

Adapter Pattern

- We know that there is a nice implementation of a ***Double-ended List*** (head and tail) interface `DList` which provides the following methods:

  - `addFront (E element)`
  - `E removeFront()`
  - `E peekFront()`

  - `addBack(E element)`
  - `E RemoveBack()`
  - `E peekBack()`

Idea: Inheritance!

- We could just make **Stack** extend the List and write the additional methods by using other existing methods.

Example:

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

Pros? Cons? Discuss with your group

Idea: Inheritance!

- It is possible just make Stack extend the List

```
public Stack<E> extends DList<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-ended linked list

C: It exposes methods that are *not supposed* to be part of the Stack interface.

Inheritance is not always the right answer

- We know that there is a nice implementation of a Double-ended List interface `DList` which provides the following methods:
  - `addFront (E element)`
  - `E removeFront()`
  - `E peekFront()`
  - `addBack(E element)`
  - `E RemoveBack()`
  - `E peekBack()`
- The other methods in the List are public and accessible by anyone. But a Stack does not expose such methods! (Ex: addBack, removeBack() etc)

Adapter Design Pattern

- In software engineering, one of the classic “design patterns” is the *adapter*.
- An *adapter* is a class that “maps” from the interface of one ADT -- the one we’re trying to implement -- into the interface of another ADT that already exists.
- If we adapt an **ADT B** to implement another **ADT A**, then every method of A must be “converted” into a related call of B.

Adapter Design Pattern

- I can use a **private** List variable

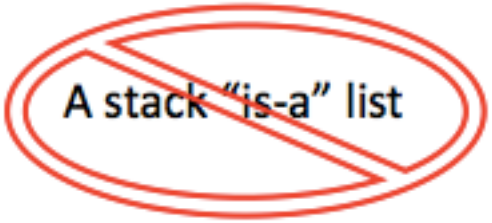
- And use its methods within a Stack class

Adapter Design Pattern

```
import java.util.*;
public class MyStack<E>
{

  private List<E> stack=new LinkedList<E>();
  public E pop()
  {
    return  stack.remove(0);
  }

  public void push (E element)
  {
    stack.add(0, element);
  }

  public static void main(String[] args)
  {
    MyStack<Integer> st=new MyStack<Integer>();
    st.push(4);
    st.push(5);
    st.pop();
    System.out.println(st.pop());
  }

}
```

Adapter Design 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.



A stack has-a list!

Adapter Design 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.
- Your class A “has a” class B.

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

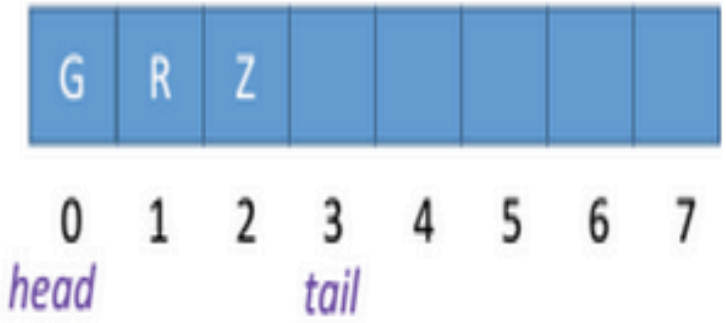
- Once this is done, we can map the methods on top of the stack to methods operating on the head of the List.

- If we choose the front....

- push -> addFront
- pop -> removeFront
- peek -> peekFront

MAPPING

Mapping between ArrayList and Stack

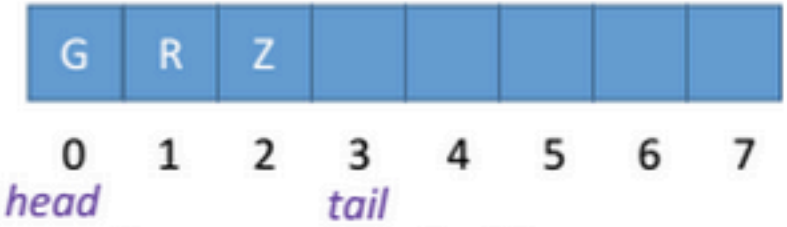


Let's say we want to adapt the ArrayList class to build our Stack implementation. In an ArrayList you can add and remove elements at either the head or the tail of the list. So which end should we choose to be the top of our Stack?

Functionally, it does not matter

But if we consider running time...

Mapping between ArrayList and Stack



- 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. $O(n^2)$, $O(n^2)$

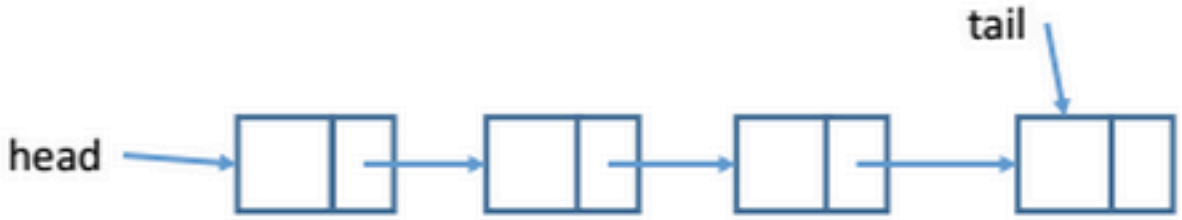
E. Other/none/more

Mapping between single linked list and Stack

- 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

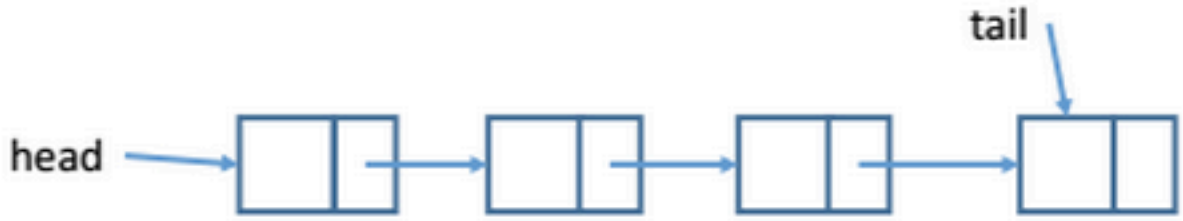
Mapping between single linked list and Stack



- 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

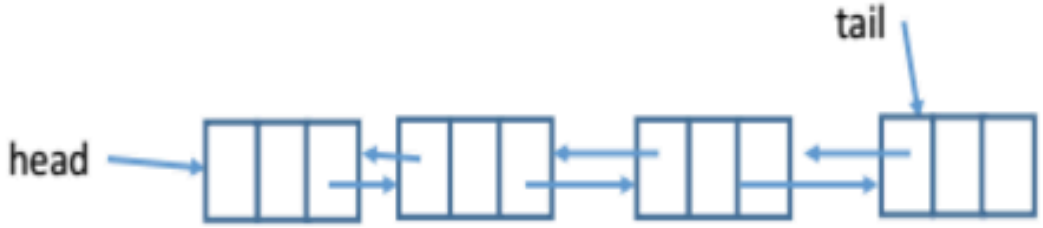
Mapping between single linked list and Stack



- 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

Mapping between a LinkedList and Stack (Stack "has a" LinkedList)



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

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

`Stack` Attribute	**`ArrayList` Equivalent**
top	size() – 1
size	size()

Don't underestimate the importance of doing this mapping first. Planning now saves time later

Map `Stack` Methods to `List` Methods

- A consequence of that attribute mapping is that a push operation results in adding to the *tail* of the List

size() – 1 + 1 = size()

tail of list | next position beyond *tail* | location to "push" the new stack element

`Stack` operation	**`List` operation equivalent**
`push( element )`	`add( size(), element )`
`E pop()`	`E remove( size() - 1 )`
`E peek()`	`E get( size() - 1 )`
`int size()`	`int size()`
`boolean isEmpty()`	`boolean isEmpty()`

Stack Implementations, done correctly…

Operation	LinkedList (head = TOS)	ArrayList (end = TOS)
Push(E element)	O(1)	O(1)
Pop()	O(1)	O(1)
Peek()	O(1)	O(1)

Famous Stack Application

- Parentheses checker algorithm