CS 228: Introduction to Data Structures
Lecture 23
Wednesday, October 19, 2016

Time Complexity (Continued)

Let us now consider the `ListIterator` methods.

`add(item)`, `remove()`: *Adding or removing an element during iteration*

- linked list: O(1)
  - Reason: Already have the node, so linking or unlinking is O(1)
- array list: O(n)
  - Reason: Have to shift elements to add/remove

Given a list of length n, iterate over the list and perform k adds or removes at arbitrary locations:

- linked list: O(n) + O(k) = O(n + k) or O(max(n, k))
- array list: O(nk)
  - Reason: each of the k operations is potentially O(n)

Note: The last item probably sums up the potential advantage of linked lists in terms of time complexity: In this application the linked list takes linear time, and the array list takes quadratic time.

Other considerations:

- **Space**. Array lists potentially waste some space because of the empty cells. (But remember, each empty cell is just an object reference – it takes up 4 bytes, not the space of the object itself!) For many short lists, linked lists may be more space-efficient. On the other hand, linked lists potentially waste space, because each element has to be wrapped in its own node object.

- **Practical performance.** In practice, array-based operations are very fast. Adding a new node to a linked list requires creation of the node object and several memory operations. Thus `ArrayLists` tend to be good for most purposes.

So, linked data structures are interesting for lists, and, as we'll soon see, they are also useful for queues and stacks.

Later, we will see that they are absolutely indispensable for more complex structures like trees and graphs.

Stacks

A ***stack*** is an access-restricted list. You may manipulate only the item at the top of the stack: You may

- ***push*** a new item onto the top of the stack,
- ***pop*** the top item off the stack, or
- ***peek*** at (examine) the ***top*** item of the stack.

Thus, the access policy for stacks is ***last-in first-out***.

Formally, the following methods characterize a stack consisting of elements of type `E`:

`void push(E item):` Adds an element to the top of stack.

`E pop():` Removes and returns the top element of the stack. Throws `NoSuchElementException` if the stack is empty.

`E peek()`: Returns the top element of the stack without removing it. Throws `NoSuchElementException` if the stack is empty.

`boolean isEmpty()`: Return `true` if the stack is empty, `false` otherwise.

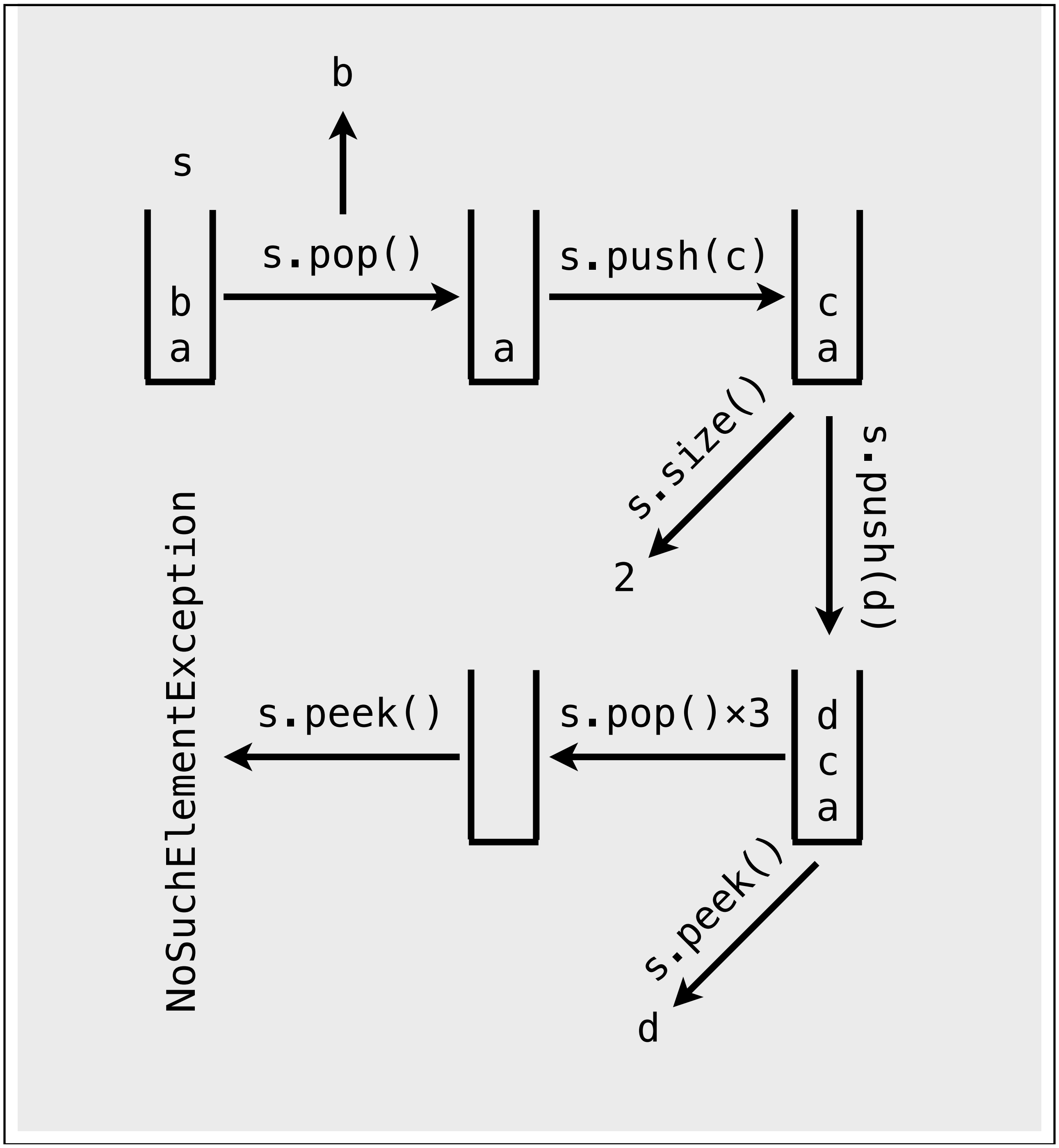
`int size()`: Returns the number of elements in the stack.

All of these methods are in the `PureStack` interface, a simple stack interface, which is posted on Blackboard.

```
public interface PureStack<E>
{
  void push(E item);

  E pop();

  E peek();

  boolean isEmpty();

  int size();
}
```

We don’t *need* iterators, although there are cases where they can be useful. In principle, but not in practice, a stack can grow arbitrarily large.

Example.

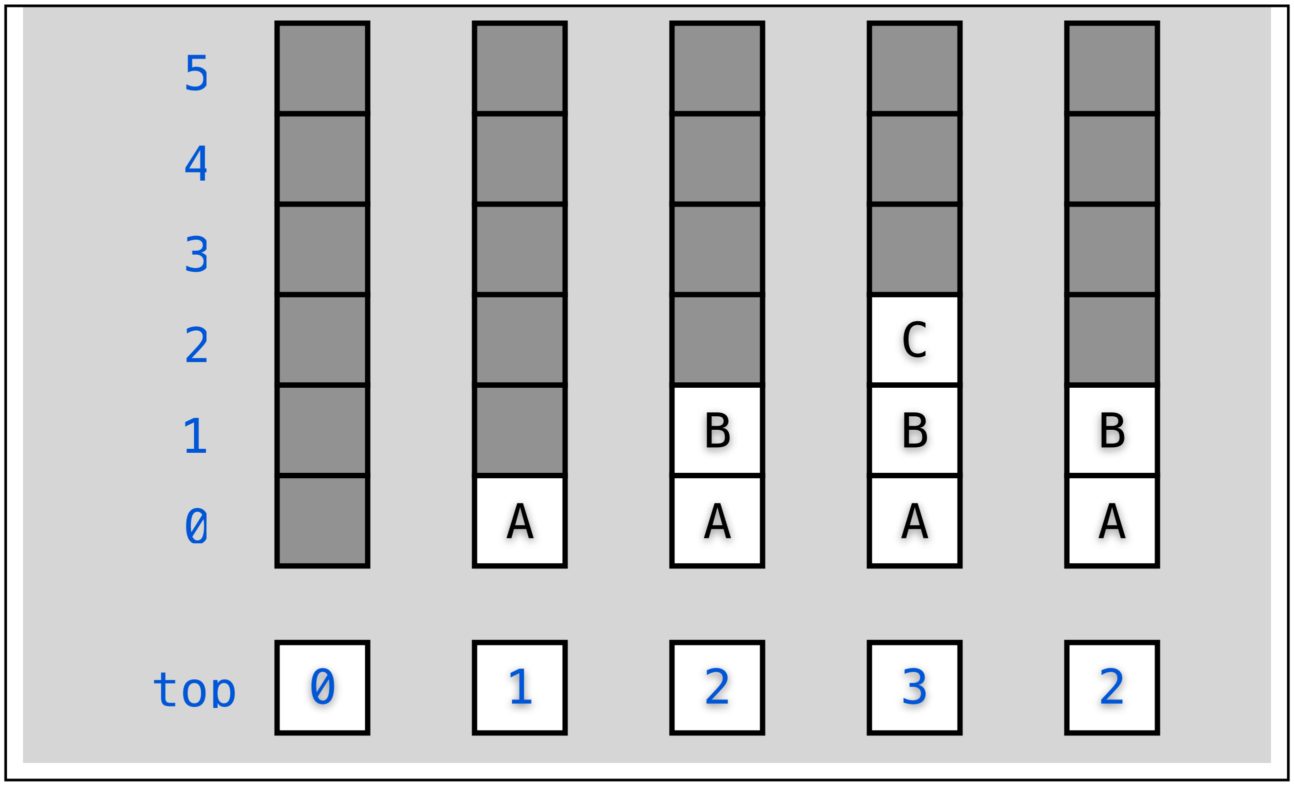


Remark. Why study stacks when we already have lists? One reason is to be able to carry on discussions with other programmers. If somebody tells you that an algorithm uses a stack, this can give you a good hint about how the algorithm works. In fact, stacks are central to Java: The Java Virtual Machine, which executes Java bytecode, is stack-based.

Direct Implementations of Stacks

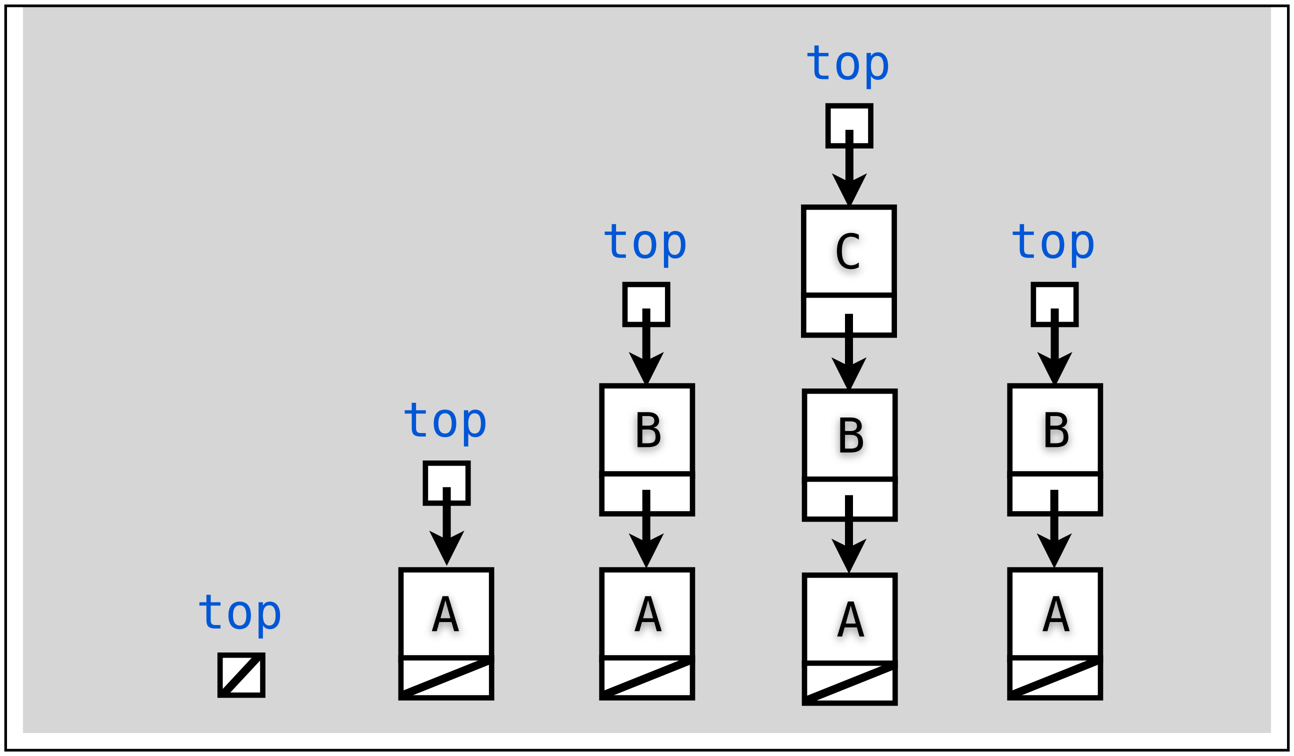
It is easy to implement stacks using arrays or linked lists. We sketch both approaches next.

Array implementation. We use a `data` array, and an index `top` into `data`. Entries `data[0]`, ... , `data[top−1]` contain the elements of the stack. A sequence of pushes and pops, starting from an empty stack might look like this.



When there is no more space in the `data` array for another `push`, just double the size of the array. All operations take O(1) time (amortized, in the case of `push`).

Linked list implementation. Singly-linked lists work well for stacks, since we only need access to the top. Simply use a sequence of linked nodes, with a pointer `top` to the first node, which is viewed as the top of the stack. This is illustrated next.



All operations take O(1) time.

Implementations

Code for the array and linked list implementations of the `PureStack` interface is posted on Blackboard — see `ArrayBasedStack` and `LinkedStack`, respectively. All operations take O(1) time (amortized, in the case of `push` for `ArrayBasedStack`).

Java Implementations of Stacks

It is easy to implement a stack as a Java `List`:

Stack Method	**`List` method**
`push()`	`add()`
`peek()`	`get(size()-1)`
`pop()`	`remove(size()-1)`
`isEmpty()`	`isEmpty()`
`size()`	`size()`

We can then use one of Java's implementations of the `List` interface. Two convenient implementations are the following.

- `ArrayList` implements `List` as a resizable array, so all the stack methods run in O(1) time. (To be precise, `add()` runs in O(1) amortized time.)

- `LinkedList` implements `List` as a doubly-linked list, so all stack methods take O(1) time.

Java has a legacy `Stack` class that implements all the required methods. However, Oracle recommends using the more modern `Deque` (for "doubly-ended queue) interface instead, as it provides "a more complete and consistent set of LIFO stack operations".

Stack Method	Deque Method
`push()`	`addFirst()`
`pop()`	`removeFirst()`
`peek()`	`peekFirst()`

`Deque` has many other methods. We will revisit this interface when we study queues. `ArrayDeque` and `LinkedList` implement `Deque`.

Note. Although the Java implementations of stacks we have seen are fine for many applications, they do come loaded with features — e.g., `indexOf()` and `listIterator()` — that may be unnecessary for your specific application. In such cases, a simple implementation, like our PureStack and, e.g., `LinkedStack` might be more suitable.

Applications of Stacks

Stacks are among the most versatile data structures in computer science. For example, as mentioned before, they are essential to the implementation of the Java Virtual Machine. In the next few lectures, will see some of the applications of stacks.

Verifying Matched Parentheses

As a first application of stacks, consider the problem of verifying whether an string of parentheses is well-formed. For instance,

“`{[(){[]}]()}`”

is well-formed, but

“`{[]}[]]()}`”

is not.

More precisely, a string γ of parentheses is well-formed if either γ is empty or γ has the form

$$(\alpha)\beta \qquad [\alpha]\beta \quad \text{or} \qquad \{\alpha\}\beta$$

where α and β are themselves well-formed strings of parentheses. This kind of recursive definition lends itself naturally to a stack-based algorithm.

To check a `String` like `"{[(){[]}]()}"`, scan it character by character.

- When you encounter a left paren — `'{'`, `'['`, or `'('` — push it onto the stack.

- When you encounter a right paren, pop its counterpart from atop the stack, and check that they match.

If there is a mismatch or exception, or if the stack is not empty when you reach the end of the string, the parentheses are not properly matched.

Detailed code is given in `ParenExample.java` on BlackBoard. It uses the `PureStack` interface, but it could have easily been done using `List` or `Deque`.