CSE 373: Data Structures and Algorithms

Lecture 1: Introduction; ADTs; Stacks

Course objectives

- Learn basic data structures and algorithms
  - data structures – how data is organized
  - algorithms – unambiguous sequence of steps to compute something
  - algorithm analysis – determining how long an algorithm will take to solve a problem
- Become a better software developer
  - "Data Structures + Algorithms = Programs"
    -- Niklaus Wirth, author of Pascal language

Abstract Data Types

- **abstract data type (ADT)**: A specification of a collection of data and the operations that can be performed on it.
  - Describes *what* a collection does, not *how* it does it
  - Described in Java with interfaces (e.g., `List`, `Map`, `Set`)
  - Separate from **implementation**

- ADTs can be implemented in multiple ways by classes:
  - `ArrayList` and `LinkedList` implement `List`
  - `HashSet` and `TreeSet` implement `Set`
  - `LinkedList`, `ArrayDeque`, etc. implement `Queue`
    - They messed up on Stack; there's no Stack interface, just a class.

`List` ADT

- An ordered collection the form A_0, A_1, ..., A_{N-1}, where N is the size of the list
- Operations described in Java's `List` interface (subset):

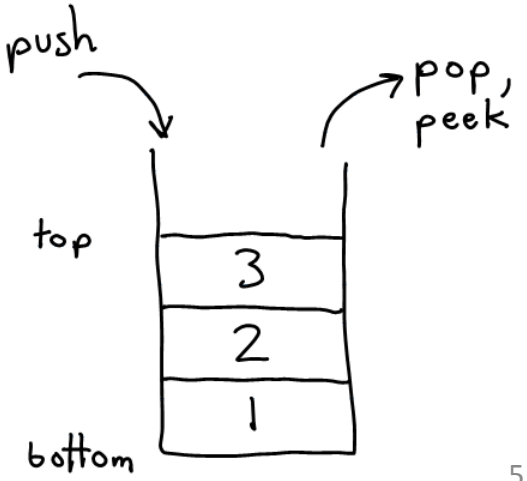
`add(`**el**`, `**index**`)`	inserts the element at the specified position in the list
`remove(`**index**`)`	removes the element at the specified position
`get(`**index**`)`	returns the element at the specified position
`set(`**index**`, `**el**`)`	replaces the element at the specified position with the specified element
`contains(`**el**`)`	returns true if the list contains the element
`size()`	returns the number of elements in the list

- `ArrayList` and `LinkedList` are implementations

Stack ADT

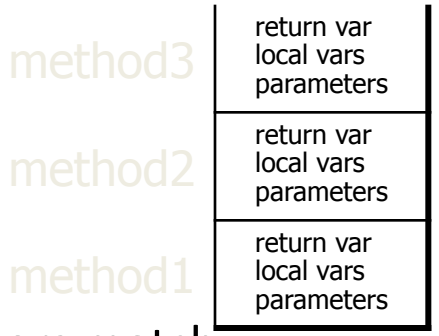
- **stack**: a list with the restriction that insertions/deletions can only be performed at the top/end of the list
  - Last-In, First-Out ("LIFO")
  - The elements are stored in order of insertion, but we do not think of them as having indexes.
  - The client can only add/remove/examine the last element added (the "top").

- basic stack operations:
  - **push**: Add an element to the top.
  - **pop**: Remove the top element.
  - **peek**: Examine the top element.



Applications of Stacks

- Programming languages and compilers:
  - method calls are placed onto a stack *(call=push, return=pop)*
  - compilers use stacks to evaluate expressions



- Matching up related pairs of things:
  - find out whether a string is a palindrome
  - examine a file to see if its braces { } and other operators match
  - convert "infix" expressions to "postfix" or "prefix"

- Sophisticated algorithms:
  - searching through a maze with "backtracking"
  - many programs use an "undo stack" of previous operations

Class `Stack`

`Stack<`**E**`>()`	constructs a new stack with elements of type **E**
`push(`**value**`)`	places given value on top of stack
`pop()`	removes top value from stack and returns it; throws `EmptyStackException` if stack is empty
`peek()`	returns top value from stack without removing it; throws `EmptyStackException` if stack is empty
`size()`	returns number of elements in stack
`isEmpty()`	returns `true` if stack has no elements

```
Stack<Integer> s = new Stack<Integer>();
s.push(42);
s.push(-3);
s.push(17);                    // bottom [42, -3, 17] top

System.out.println(s.pop()); // 17
```

Stack limitations/idioms

- Remember: You cannot loop over a stack like you do a list.

```
Stack<Integer> s = new Stack<Integer>();
…
for (int i = 0; i < s.size(); i++) {
    do something with s.get(i);
}
```

- Instead, you pull contents out of the stack to view them.
  - common idiom: Remove each element until the stack is empty.

```
while (!s.isEmpty()) {
    do something with  s.pop();
}
```

Exercise

- Write a method `symbolsBalanced` that accepts a `String` as a parameter and returns whether or not the parentheses and the curly brackets in that `String` are balanced as they would have to be in a valid Java program.
  - Use a Stack to solve this problem.