Lecture 3: Abstract Data Types (ADT): Collections, Lists, Stacks and Queues

Reading materials
Dale, Joyce, Weems: 2.1-2.3, 3.1, 3.5-3.6, 5.1-5.5 (for now ignore mentions of generics and writing exception classes)
OpenDSA: 4.1, 5.1-5.3,
Liang: 10.9, Comprehensive: 22

Topics Covered

1 Abstract Data Types 2
2 Collection 2
3 List ADT 2
   3.1 StringList ADT: Informal Specification 3
   3.2 StringList ADT: Formal Specification 3
4 Stacks ADT 4
   4.1 CharStack ADT: Informal Specification 4
   4.2 CharStack ADT: Formal Specification 4
5 Queue ADT 4
   5.1 ProcessQueue ADT: Informal Specification 4
   5.2 PrintJobQueueADT: Formal Specification 5
6 Prefix and Postfix Expressions 5
   6.1 Prefix Notation 5
   6.2 Postfix Notation 6
   6.3 Today’s Uses of Postfix Notation 7
1 Abstract Data Types

What is the abstraction? Abstraction is a model of a system that includes only the details needed by the viewer/user of such system. The complexity and details of actual implementation should be hidden from the user - that is called information hiding. Why? Because the complex design details are not relevant to the user and may make it actually harder to understand the system for the user. Imagine what would happen if you needed to know and understand every detail of how cars work, before you could use them.

An abstract data type (ADT) provides a collection of data and a set of operations that act on the data. An ADT’s operations can be used without knowing their implementations or how the data is stored, as long as the interface to the ADT is precisely specified. An ADT is implementation independent and, in fact, can be implemented in many different ways (and many different programming languages).

In this course we will use two different ways of specifying an ADT:

- **informal specification** - an English description that provides list of all available operations on the data with their inputs and outputs;
- **formal specification** - a Java interface description that later can be implemented by concrete classes.

We will be using ADTs from three different perspectives:

- **application level** (or user, or client): using a class for which you only know the formal ADT specification;
- **logical level**: designing the ADT itself given a set of requirements specified by the non-programmer user (this may involve asking questions);
- **implementation level**: writing code for a class that implements a given ADT.

2 Collection

A collection object is an object that holds other objects. Typical operations provided by collection classes are:

- insert,
- remove,
- iterate through the collection.

3 List ADT

There are many different possible List abstract data types that require different sets of operations to be defined on them. The ADT for the list that we define in this lecture is a very general one. We will use it (after slight revisions) in several future lectures and provide different ways of implementing the list interface.

For now, we will specify List ADTs that expect objects of a specific type. Once we discuss Java generics, we will revise the List ADT to be general enough to work with any chosen type of objects.
3.1 StringList ADT: Informal Specification

The StringList contains a (possibly empty) collection of objects of type String. The list supports the following operations:

- **insert**: This operation adds a String object, given as a parameter, to the list of strings.
- **remove**: This operation removes a String object, given as a parameter, from the list of strings. If given String object is not on the list, the list content does not change.
- **clear**: This operation removes all objects from the list. The list becomes empty.
- **contains**: This operation determines if a String object, given as a parameter, is stored in the list or not.
- **indexOf**: This operation determines the index (or location) of a String object, given as a parameter. It returns -1 if the given String object is not stored in the list. (The indeces do not imply any ordering of the objects in the list and may have different value depending on the implementation of this ADT).
- **get**: This operation returns an object stored at the index/location given as a parameter.
- **size**: This operation determines the number of objects stored in the list.
- **toString**: This operation produces a meaningful String representation of the list.

Note that this particular StringList ADT differs significantly from the StringLog ADT provided in chapter 2 of [Dale, Joyce, Weems] and from the List ADT provided in chapter 5 of OpenDSA book. This does not make any of these ADTs incorrect or better or worse than the others; they are simply different ADTs.

3.2 StringList ADT: Formal Specification

The formal specification of the ADT is given by Java’s `interface`. This makes it language specific, but it is still general enough to be adaptable by implementations in other programming languages.

```java
public interface StringListInterface {
    void insert (String item);
    void remove (String item);
    void clear ();
    boolean contains (String item);
    int indexOf (String item);
    String get (int index);
    int size ();
    String toString ();
}
```

See the `StringList.java` file in the lecture02 source code for a documented version of the above interface.

See List interface defined in Java libraries, http://docs.oracle.com/javase/7/docs/api/java/util/List.html, for an example of similar interface.
4 Stacks ADT

Stacks are structures in which elements are always added and removed from the same end (depending on how you visualize the stack, you may wish to think of that end as the *top* of the stack). Stacks are *last in first out* (or LIFO) structures.

4.1 CharStack ADT: Informal Specification

The CharStack contains a (possibly empty) collection of objects of type Character. The stack supports the following operations:

**insert / push** This operation adds a Character object, given as a parameter, to the top of the stack of characters.

**remove / pop** This operation removes and returns a Character object from the top of the stack of characters.

**peak** This operation returns a Character object from the top of the stack of characters.

**toString** This operation produces a meaningful String representation of the stack.

4.2 CharStack ADT: Formal Specification

```java
public interface CharStack {
    public void push (Character item);
    public Character pop();
    public Character peak();
    public String toString();
}
```

See the CharStack.java file in lecture source code for a documented version of the above interface.

5 Queue ADT

Queues are structures in which elements are added to one end (rear/back of a queue) and removed from the other end (front of a queue). Queues are *first in first out* structures (FIFO).

5.1 ProcessQueue ADT: Informal Specification

The ProcessStack contains a (possibly empty) collection of objects of type Process. Process is a user defined class representing a process waiting to be scheduled to run on the CPU of a computer. The queue supports the following operations:

**insert / enqueue** This operation adds a Process object, given as a parameter, to the end of the queue of processes.

**remove / dequeue** This operation removes and returns a Process object from the front of the queue of processes.

**toString** This operation produces a meaningful String representation of the stack.
### 5.2 PrintJobQueueADT: Formal Specification

```java
public interface PrintJobQueue {
    public void enqueue ( PrintJob item );
    public PrintJob dequeue ( ) ;
    public String toString ( ) ;
}
```

See the `PrintJobQueue.java` file in lecture source code for a documented version of the above interface.

### 6 Prefix and Postfix Expressions

How would you write a program to evaluate a mathematical, arithmetic expression? For example something like this

\[
((15/(7-(1+1))) \times 3)-(2+(1+1)).
\]

The code that can evaluate such expressions has to:

- find and evaluate all subexpressions that are surrounded by parenthesis;
- for each operator figure out what its operands are; this involves determining precedence of operators (i.e. knowing the order of operations).

It turns out that it is much simpler to write code for evaluation of arithmetic expressions when they are written in a format that lets us ignore parenthesis and that does not depend on operator precedence. These two ways are prefix and postfix notations.

#### 6.1 Prefix Notation

**Prefix notation** is a notation for writing arithmetic expressions in which the operator comes before its operands.

<table>
<thead>
<tr>
<th>infix</th>
<th>postfix</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 + 5</td>
<td>+ 2 5</td>
</tr>
<tr>
<td>(2 + 4) \times 5</td>
<td>\times + 2 4 5</td>
</tr>
<tr>
<td>2 + 4 \times 5</td>
<td>+ 2 \times 4 5</td>
</tr>
<tr>
<td>((15/(7-(1+1))) \times 3)-(2+(1+1))</td>
<td>- \times / 15 - 7 + 1 1 3 + 2 + 1 1</td>
</tr>
</tbody>
</table>

**Algorithm for evaluating prefix expressions**

1. Scan the given prefix expression from right to left
2. For each symbol
if operand then
   push onto stack
else if operator then
   operand1 = pop stack
   operand2 = pop stack
   compute operand1 operator operand2
   push result onto stack

return top of stack as result

6.2 Postfix Notation

Postfix notation is a notation for writing arithmetic expressions in which the operator follows its operands.

<table>
<thead>
<tr>
<th>infix</th>
<th>postfix</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2 + 5$</td>
<td>$2 \ 5 \ +$</td>
</tr>
<tr>
<td>$(2 + 4) \times 5$</td>
<td>$2 \ 4 \ 5 \times$</td>
</tr>
<tr>
<td>$2 + 4 \times 5$</td>
<td>$2 \ 4 \ 5 \times \ +$</td>
</tr>
<tr>
<td>$((15/(7-(1+1))) \times 3)-(2+(1+1))$</td>
<td>$15 \ 7 \ 1 \ 1 \ + \ - \ / \ 3 \ + \ 2 \ 1 \ 1 \ + \ + \ -$</td>
</tr>
</tbody>
</table>

Algorithm for evaluating postfix expressions

for each character in the input postfix string expression
   if the character is an operand
      push it (its value) onto a stack
   else if the characted is an operator
      operand2 = pop stack
      operand1 = pop stack
      compute operand1 operator operand2
      push result onto stack

return top of the stack as result

Algorithm for converting infix to postfix expressions

for each character in the input infix string expression
   if the character is an operand
      append to postfix string expression
   else if the character is a left brace
      push it onto the operator stack
   else if the character is an operator
      if the stack is not empty
         while top element on the stack has higher precedence
            compute operand1 operator operand2
            push result onto stack
         else if the character is a left brace
            push it onto the operator stack
      else if the character is a right brace
         while top element on the stack is not a left brace
            compute operand1 operator operand2
            push result onto stack

return top of stack as result
pop the stack and append to postfix string expression
push it (the current operator) onto the operator stack
else if the character is a right brace
while the operator stack is not empty
  if the top of the operator stack is not a matching left brace
    pop the operator stack and append to postfix string expression
  else
    pop the left brace and discard
    break
while the operator stack is not empty
  pop the operator stack and append to postfix string expression
operators on the stack.

6.3 Today’s Uses of Postfix Notation

RPN stands for "Reverse Polish Notation" which is another name for postfix notation.

Many of the HP Calculators still use RPN as well as infix notation.