7 Introduction to ADTs and Data Structures
Chapter 14

Previously in 415-416

- Data structures
  - basic Java arrays
- Abstract Data Types
  - ArrayList, Vector

Preview

- Specification versus implementation
  - Abstract Data Types (ADT): specification
  - data structure: implementation
- ADT examples
- Implementation examples
- Using ADTs

Specification vs Implementation

- abstract data type (ADT): A specification of a collection of data and the operations that can be performed on it.
- Specification of an Abstract Data Type class
  - Public interface that defines the *access* to the data and the associated *semantics (behevior of data)*
- Implementation of an ADT class
  - Should be hidden
  - Should be able to change it without changing ADT
  - Typically uses a *concrete data structure*

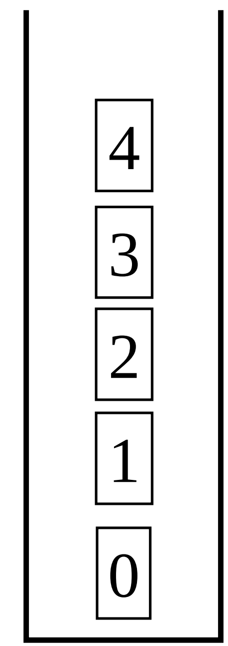
Linear ADTs

- An ADT that *models* its data members as a <u>sequence </u>is called a *<u>linear abstract data type</u>*
  - Examples: array, list, stack, queue, dequeue, et al.
- *Creation* options
  - *add* element: where? by position? by value?
  - *delete* element: which? by position? by value?
- *Access* options:
  - <u>sequential</u>: *first()*, *next()*
  - <u>random</u>: *get*(byIndex), *get*(byValue)

Stack

- A collection based on the principle of add singe elements and retrieving them in the opposite order.
  - Last-In, First-Out ("LIFO")
  - Elements are stored in order of insertion. We do not think of them as having indexes.
- 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.

Push 0
Push 1
Push 2
Pop
Push 3
Push 4
Pop
Pop
Pop
Pop



pop order: | 2 | 4 | 3 | 1 | 0 |

Stacks in computer science

- 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 { } match
  - convert "infix" expressions to pre/postfix
- Sophisticated algorithms:
  - searching through a maze with "backtracking"
  - many programs use an "undo stack" of previous operations

Exception Stack Trace

```
Exception in thread "main" java.lang.NullPointerException
  at Swarm.Bee.<init>(Bee.java:14)
  at Swarm.Swarm.<init>(Swarm.java:36)
  at Swarm.App.<init>(App.java:30)
  at Swarm.App.main(App.java:48)
```

- At the top of the stack trace is the method that threw the Exception at runtime
  - Below is the method that called that method
  - and so on and so on..

Programming with 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

Example

```
import java.util.*;

public class StackDemo
{
  public static void main( String[] args )
  {
    Stack<Integer> iStack = new Stack<Integer>();

    iStack.push(5);
    iStack.push(36);
    iStack.push(18);

    System.out.println(iStack);
    while ( !iStack.empty() )
    {
      System.out.println(iStack.pop());
    }
  }
}
```

More Example ... but

```
import java.util.*;

public class StackDemo2
{
  public static void main( String[] args ){
    Stack<Integer> iStack = new Stack<Integer>();
    iStack.push(5);
    iStack.push(36);
    iStack.push(18);
    printStack(iStack);
  }

  public static void printStack(Stack<Integer> iStack){
    while ( !iStack.empty() )
    {
      System.out.println(iStack.pop());
    }
  }
}
```

Stack ADT

- Add element: *push(Element)*
- Remove element: *Element pop()*
- Utility behavior
  - Test if there is something to pop: *isEmpty()*
    - *pop* of empty stack can return *null* or throw exception
  - If stack has fixed size may have *isFull()*
    - *push* on a full stack should result in an exception
  - Might want: *int size()*
  - Might have “read” without removal: *Element peek()*

A String Stack

- We can easily create a *Stack* of *String* using a *Vector*

```
public class Stack
{
   private Vector<String> _stack;
   public Stack()
   {
      _stack = new Vector<String>();
   }
   public void push( String item )
   {
      _stack.add( 0, item );
   }
   public String pop()
   {
      String retVal = null;
      if ( _stack.size() > 0 )
      {
         retVal = _stack.get( 0 );
         _stack.remove( 0 );
      }
      return retVal;
   }
   public boolean isEmpty()
   {
      return _stack.size() == 0;
   }
```

Add new item at position 0

pop returns item in position 0, after it is removed from the Vector

isEmpty() is trivial

A Generic Stack

Add a generic type spec

Replace every reference to `String` with a reference to `E`

```
public class Stack <E>
{
   private Vector< E > _stack;
   public Stack()
   {
      _stack = new Vector< E >();
   }
   public void push( E item )
   {
      _stack.add( 0, item );
   }
   public E pop()
   {
      E  retVal = null;
      if ( _stack.size() > 0 )
      {
         retVal = _stack.get( 0 );
         _stack.remove( 0 );
      }
      return retVal;
   }
   public boolean isEmpty()
   {
      return _stack.size() == 0;
   }
}
```

- *Vector* is an example of Java <u>generics</u>
  - can store <u>any</u> class of object
  - class specified in < … >
- We can define *Stack* so it can also store any kind of object

Using a Generic Stack

- Use a generic *Stack* like a generic *Vector*
- StackDemo reads tokens from terminal into *Integer* and *String* stacks:
  - any *int* is pushed onto an *Integer* stack
  - "ipop" and "spop" as input pops the *Integer* and *String* stacks
  - anything else is a push onto a *String* stack

```
public class StackDemo
{
  public static main( String[] args )
  {
    Stack<String>  sStack = new Stack<String>();
    Stack<Integer> iStack = new Stack<Integer>();

    Scanner in = new Scanner( System.in );
    while ( in.hasNext() )
    {
      if ( in.hasNextInt() )
        iStack.push( in.nextInt() );
      else
      {
        String token = in.next();
        if ( token.equals( "ipop" ))
           popAction( iStack.pop() );
        else if ( token.equals( "spop" ))
           popAction( sStack.pop() );
        else  // all other tokens are String push
           sStack.push( in.next() );
      }
    }
    . . . // pop everything from both stacks
  }
}
```

Stack Algorithms

- A stack is LIFO (last in, first out)
- A stack reverses input
- A stack can hold "postponed obligations"
  - When a more pressing task occurs, push it onto the stack
  - When it is popped, the most recent previous task is resumed

Match Parenthesis

Example: (a + 10* (b + 10 * d))

```
construct empty  stack
for each token in expression
   if token is (
      push (
   else if token is )
      if stack is empty
         ERROR Mismatched paren
      else
         pop stack
   else
      ignore other tokens

if stack is not empty
      ERROR Mismatched paren
```

Input	Stack
$\boxed{(}$a + 10 * (b + 10 * d))	(
($\boxed{a}$ + 10 * (b + 10 * d))	(
(a $\boxed{+}$ 10 * (b + 10 * d))	(
(a + $\boxed{10}$ * (b + 10 * d))	(
(a + 10 $\boxed{*}$ (b + 10 * d))	(
(a + 10 * $\boxed{(}$ b + 10 * d))	((
(a + 10 * ($\boxed{b}$ + 10 * d))	((
(a + 10 * (b $\boxed{+}$ 10 * d))	((
(a + 10 * (b + $\boxed{10}$ * d))	((
(a + 10 * (b + 10 $\boxed{*}$ d))	((
(a + 10 * (b + 10 * $\boxed{d}$))	((
(a + 10 * (b + 10 * d $\boxed{)}$)	(
(a + 10 * (b + 10 * d) $\boxed{)}$	

Infix Expression Notation

- We are used to writing arithmetic expressions in <u>*infix*</u> notation with the operator *between* the operands:
  - $a + b$, $a - b$, $a + b * c$, etc.
- Although intuitive, this creates complicated rules regarding the order of operator evaluation, which leads to conventions for *operator precedence* and the use of parentheses to override precedence:
  - $(a + b) * c / (d - e)$
- There <u>are</u> other (simpler) notations

Postfix Notation

- Using *postfix* notation, the operator *follows* the 2 operands to which it applies:
  - $ab+$, $ab-$, $ab*$, etc.
- An operand can be an expression:
  - $abc*+$ means: the operands for + are a and $bc*$
  - $ab+c*de-/$ ™ $(a + b) * c / (d - e)$
- Although it seems less "natural", it is much easier to evaluate:
  - no parentheses, no precedence rules

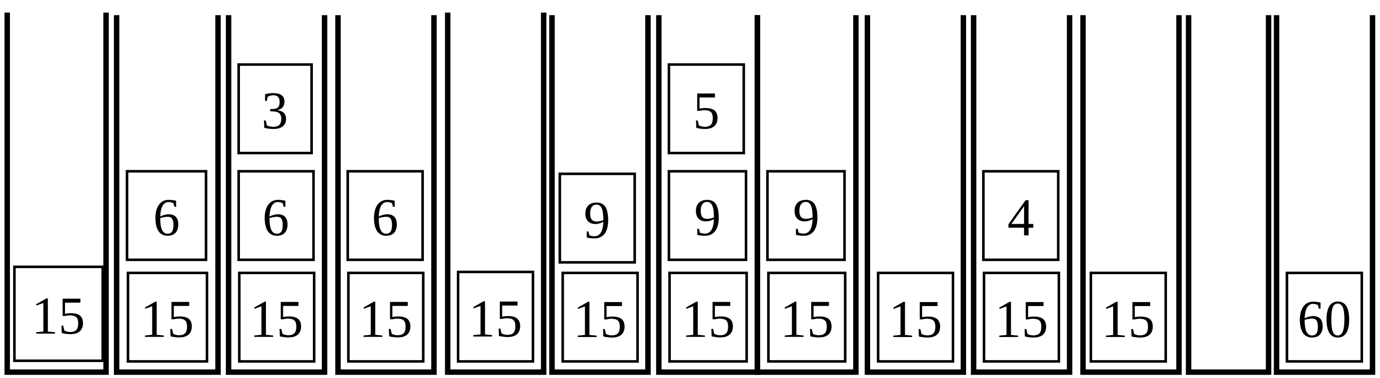
Evaluate Postfix

```
// postfix eval for binary operators
operand stack = empty
for each token in expression
  if token is operand
    push operand onto operand stack
  else // operator
    pop right operand from stack
    pop left operand from stack
    calculate left op right
    push result
//stack should have one value at end
```

Expr:

15	6	3	+	5	-	*

In	Action	L	R	V
15	Push 15			
6	Push 6			
3	Push 3			
+	Pop to R, Pop to L	6	3	
	Eval +, Push V	6	3	9
5	Push 5	6	3	9
-	Pop to R, Pop to L	9	5	9
	Eval -, Push V	9	5	4
*	Pop to R, Pop to L	15	4	4
	Eval *, Push V	15 5	4	60



Stack history

How can we extend algorithm to support operators with 1 or 3 or more operands?

Basic InFix to PostFix

(no parentheses)

```
Infix to postfix (no parens):
create empty opStack
create empty postfix list
for each token in infix
  if token is operand
    add token to postfix
  else // it is an operator
    while opStack not empty
        & prec(top) >= prec(token)
      pop opStack to postfix
    push token onto opStack

// copy remaining ops to postfix
while ( opStack !empty )
   pop opStk to postfix
```

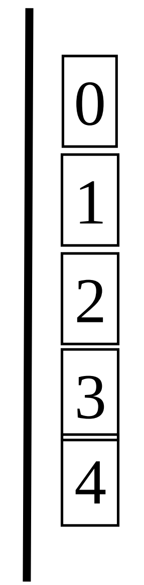
Example: a + b * c / d - e

Input	OpStack	Postfix
$\boxed{a}$ + b * c / d - e		a
a $\boxed{+}$ b * c / d - e	+	a
a + $\boxed{b}$ * c / d - e	+	a b
a + b $\boxed{*}$ c / d - e	* +	a b
a + b * $\boxed{c}$ / d - e	* +	a b c
a + b * c $\boxed{/}$ d - e	+	a b c *
	/ +	a b c *
a + b * c / $\boxed{d}$ - e	/ +	a b c * d
a + b * c / d $\boxed{-}$ e	+	a b c * d /
		a b c * d / +
	-	a b c * d / +
a + b * c / d - $\boxed{e}$	-	a b c * d / + e
a + b * c / d - e $\boxed{}$		a b c * d / + e -

Queue

- A linear collection of objects such that you can only add to one end (the back) and you can only remove from the other end (the front);
  - a *FIFO* (First In, First Out) data structure.
  - Elements are stored in order of insertion. We do not think of them as having indexes.

add 0
add 1
add 2
remove
add 3
add 4
remove
remove
remove
remove



remove order:

0	1	2	3	4

- Basic queue operation
  - add (enqueue): Add an element to the back.
  - remove (dequeue): Remove the front element.
  - peek: Examine the front element.

Programming with `Queues`

`add(`**value**`)`
  places given value at back of queue

`remove()`
  removes value from front of queue and returns it;
  throws a `NoSuchElementException` if queue is empty

`peek()`
  returns front value from queue without removing it;
  returns `null` if queue is empty

`size()`
  returns number of elements in queue

`isEmpty()`
  returns `true` if queue has no elements

Programming with `Queues`

```
Queue<Integer> q = new LinkedList<Integer>();
q.add(2);
q.add(-3);
q.add(7);          // front [2, -3, 7] back

System.out.println(q.remove());    // 2
```

IMPORTANT:

Java Queue is an interface.

When constructing a queue you must use a new LinkedList object instead of a new Queue object.

Example ... but

```
import java.util.*;
public class QueueDemo
{
  public static void main( String[] args )
  {
    Queue<Integer> iQue = new LinkedList<Integer>();
    iQue.add(5);
    iQue.add(36);
    iQue.add(18);

    System.out.println(iQue);
    printQueue(iQue);
  }
  public static void printQueue(Queue<Integer> iQueue){
    while ( !iQueue.isEmpty() )
    {
      System.out.println(iQueue.remove());
    }
  }
}
```

Queue ADT

- Add element: *add(Element)* or *enqueue*
- Remove element: *Element remove()* or *dequeue*
- Utility behavior
  - Test if there is something to remove: *isEmpty()*
    - *remove* when empty can return *null* or throw exception
  - If queue has fixed size, might have: *isFull()*
    - *add* when full results in an exception
  - Might want: *int size()*
  - Might have “read” without removal: *Element front()*

A Queue Implementation

- Based on Vector

Essentially identical to the Stack implementation, except we add to the end.

```
public class Queue <E>
{
   private Vector<E> _queue;
   public Queue()
   {
      _queue = new Vector<E>();
   }
   public void add( E item )
   {
      _queue.add( item );
   }
   public E remove()
   {
      E retVal = null;
      if ( _queue.size() > 0 )
      {
         retVal = _queue.get( 0 );
         _queue.remove( 0 );
      }
      return retVal;
   }
   public boolean isEmpty()
   {
      return _queue.size() == 0;
   }
}
```

Queue Algorithms

- Queues are FIFO (first in, first out)
- Maintain input order
- Can be used as a "buffer" between parts of a program

Queue Examples

- **Buffers** to store values for later processing where it is important to preserve original order, such as the output buffer used by the System for standard output.
  - the program places the characters to be output into the buffer, the operating system removes them from the buffer and "prints" them when convenient
- Buffers may be used to connect two parts of a program, the output of one part goes into a buffer, the second part takes the values from the buffer as its "input"
  - This is basis for Unix notion of a command line "pipe", | operator:

```
% grep "//" file.java | wc
```

The output from *grep* (a pattern matcher) is "piped" (|) to the input for *wc* (counts words and lines)

Mixing Stacks and Queues

We often can mix stacks and queues to achieve certain effects.

Example: Reverse the order of the elements in a queue.

```
Queue<Integer> q = new LinkedList<Integer>();
q.add(1);
q.add(2);
q.add(3);                          // [1, 2, 3]
Stack<Integer> s = new Stack<Integer>();
while (!q.isEmpty()) {             // Q -> S
    s.push(q.remove());
}
while (!s.isEmpty()) {             // S -> Q
    q.add(s.pop());
}
System.out.println(q);             // [3, 2, 1]
```