July 20, 2010

Collections

A collection is a data structure that holds an unspecified amount of data. This is different from an array where the size of the array is specified when you first create it. Collections always start with a size of 0.

Two types:

- Linear Collections: Maintain data in a sequential, one-dimensional line
- Non-Linear Collections: Maintain data in a network, grid, hierarchy, or no order at all.

A collection data structure should have these two operations at the very least:

- add: adds an element to the data structure, thus increasing the size by 1.
- remove: removes an element from the data structure, thus decreasing the size by 1.

Collections may have more operations, such as a way to access individual elements.

Goals of a collections:

- Abstraction: Hide the internal details of how it works. There are two different ways of building a stack, and the user should not be aware of either of them.
- Encapsulation: Protect the internal data from tampering and restrict how the data is inserted and accessed.

For every collection, we ask the following:

- How will the collection operate?
- How will the interface look?
- What problems will the collection solve?
- How will we implement the collection?
- What are the benefits and costs of each implementation?

Generics

Often times we need to make a library of data to hold data in a particular form. Many times we don't care what that form is because the data itself isn't relevant to the problem. The linked list is a good example of this. It stores and releases data regardless of the type of data. In the homework assignment, we use the type “int” for convenience. When we define a class, if we add _<SomeType>_ out to the side, then the user of our library can specify what they want _<SomeType>_ to be. The drawback to generics is that it only works for objects. You can't specify the generic type to be a primitive. The upside to this is that each of the primitive data types have an analogous Object type. “int” is “Integer”, “double” is “Double”, etc. You can make the name of _SomeType_ to be whatever you want as long at the name doesn't conflict with another class or type currently in Java. Typically, I just use ”<T>” (which is short for “type”) to be my generic data type. Through out the textbook, the authors use ”<E>” to represent a generic type. I fail to understand the author's reasoning behind this naming convention.

```
class MyClass<T> {

    T field; // The variable "field" can now be any data type that the user desires.

}
```

To instantiate a class that uses generics, it needs to be declared like this:

```
MyClass<Integer> m = new MyClass<Integer>();
```

Stacks

There are some data structures that limit how you access the data. A stack is a common data structure for collecting data as needed. The stack allows you to add as much data to it as you like, but you only have access to the most recent value. We call the stack a Last-In-First-Out (LIFO) structure. The last value to get thrown on the stack is the first value to get removed from the stack.

Think of a stack as a literal stack of plates on a buffet bar in a restaurant. The person eating will walk up to the bar, grab the top plate, and starts to add food to their plate. When the stack gets low, someone from the kitchen will come out and add clean plates to the stack. The first plate to touch the stack is on bottom, so it is grabbed last. The top plate is still always grabbed first.

The back button on your web browser is also an example of a stack. Every time you go to a page, the last page you were at gets added onto the stack. The top of the stack is always the last page you visited. By hitting the back button, you are removing the last page from the stack and making it the current page.

The "undo" button on your text editor will undo your last action on a file. Every action you perform gets added onto a stack. The top of the stack is always your latest action. When you click "undo", you are removing the top action from the stack, making the previous action the new top of the stack.

There are six stack operations:

- boolean isEmpty() - Returns true if stack is empty, false otherwise
- boolean isFull() - Returns true if stack is full, false otherwise
- int size() - Returns the size of the stack
- void push(T data) - Adds a value to the top of the stack (increasing size by 1)
- T pop() - Removes a value from the top of the stack and returns it (decreasing size by 1)
- T top() - Returns the top of the stack

Infix, Polish, and Reverse Polish Notation

When we write a math program, we usually use what's called "infix" notation. The best example of this is "2 + 2". We call it "infix" because the "+" appears between "2" and "2". Think of "2 + 2" as a Java method. The "+" is the method name. We'll call this method "add". "add" requires two parameters. We will call "add" in this manner: "add(2, 2)". We can rewrite the four basic operations as Java methods:

- int add(int a, int b) { return a + b; }
- int subtract(int a, int b) { return a - b; }
- int multiply(int a, int b) { return a * b; }
- int divide(int n, int d) { return n / d; }

Infix notation has a drawback. It requires you to memorize an order of operations, which include parentheses to override certain operations.

```
(2 + 2) * 9          // Infix notation

multiply(add(2,2), 9) // Java methods
```

Look at the "Java methods" version. If we replace the words "multiply" and "add" with their mathematical symbols, we get "Polish" notation:

```
* + 2 2 9            // Polish notation

*
```

There is no need for parentheses or order of operations to understand which actions take place first. Someone (much smarter than me) realized a very interesting property if we take Polish notation and reverse where the operations symbols go. Instead of before the operation, make it after the operation.

```
((2,2)add, 9)multiply // Reverse Java Methods (This isn't legal, but I'm using this to demonstrate what it might look like.)

2 2 + 9 *            // Reverse Polish Notation
```

How to "parse" and solve a Reverse Polish Notation expression.

```
method RPN

    stack <- new stack
```

```
        WHILE nextToken
            token <- getNextToken
            if token is NUMBER
                PUSH stack token
            else if token is OPERATOR
                a <- POP stack
                b <- POP stack
                c <- a (operation of token) b
                PUSH stack c
            end if
        end while
        PRINT TOP stack
    end method
```

If everything works, at the end of the loop after all of the tokens are read, there should be only one value on the stack, and that will be the result of the solution of the expression.

Parsing strings from an expression. This method will match strings of parentheses.

```
    method matchParens
        stack <- new stack
        WHILE nextToken
            token <- getNextToken
            if token is '(' OR token is '{' OR token is '['
                PUSH stack token
            else if token is ')' OR token is '}' OR token is ']'
                if IS_EMPTY stack
                    return false
                end if

                if (TOP stack is '(' AND token is ')') OR (TOP stack is '{' AND token is '}') OR (TOP stack is '[' AND token is ']')
                    POP stack
                else
                    return false
                end if
            end if
        end while
        return IS_EMPTY stack
    end method
```

Stack Implementation

```
class Stack


    MAX_SIZE <- 100


    T[] elements <- Array of length MAX_SIZE
    top <- 0


    boolean isEmpty()
        return (top == 0)
    end isEmpty


    boolean isFull()
        return (top == MAX_SIZE)
    end isFull


    int size()
        return top
    end size


    void push(T data)
        if isFull()
            print "ERROR: Cannot push item on stack because stack is full."
            exit
        end if


        elements[top] <- t
        top <- top + 1
    end push


    int pop()
        if isEmpty()
            print "ERROR: Cannot pop item from stack because there is nothing to pop"
            exit
        end if


        top <- top - 1
        return elements[top]
        end pop

        T top()
            if isEmpty()
                print "Error: Cannot report top of stack because stack is empty."
                exit
            end if

            return elements[top-1]
        end top

    end Stack
```