Lecture 11: Stacks, Queues, and Linked Lists *built on 2019/05/27 at 21:04:06*

We will now develop a few data structures that support common data-access patterns such as last-in first-out (LIFO), first-in first-out (FIFO), and a generic linear data structure that maintains a list.

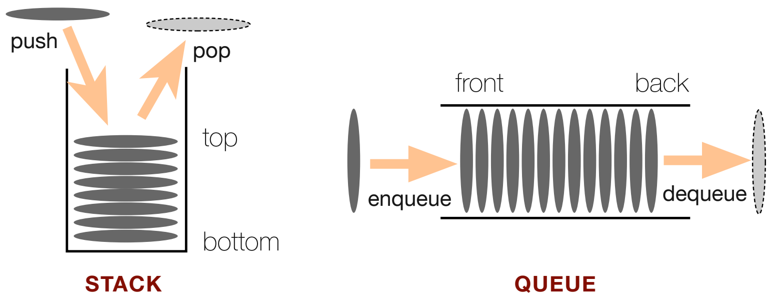
1 First-In First-Out, Last-In First-Out

Stacks and queues are fundamental data types that maintain a collection of objects supporting operations such as adding a new item, deleting an item (not arbitrary), and testing if the collection is empty. In both data types, adding an item means simply adding an extra item to the collection.

The distinguishing trait is which item to remove when a delete operation is called:

- a stack removes the most recently-added item (hence LIFO), analogous to the back button of your browser or a pile of plates (which more or less can only be removed from the top);
- by contrast, a queue removes the least recently added item (hence FIFO), analogous to a line formed to order food.

For historical reasons, the add and delete operations for stacks and queues have different names. Their names and actions are depicted below:



2 Stacks

In the most basic form, the *stack* data type is a basic collection that supports the following operations:

- `new()` — make a new Stack instance (initially empty).
- `push(elt)` — add to the collection the item `elt`.
- `pop()` — remove and return the item at the top of the stack (i.e., the item that was most recently added).
- `top()` — return (without removing) the item at the top of the stack.
- `isEmpty()` — return a Boolean indicating whether the stack is empty.

This means we can expect the following interface in Java:

```java
public interface Stack<T> {
   // add elt to the top of the stack
   public void push(T elt);

   // remove and return the item at the top of the stack (most-recently added).
   public T pop();

   // return without removing the item at the top of the stack.
   public T top();

   // return a boolean indicating whether the stack is empty
```

```
  public boolean isEmpty();
}
```

Despite the simplicity, this data type has many (real-world) applications, including

- code parsing in compilers
- interpreter for the PostScript language for professional-grade printers
- how function calls are mostly implemented.

We will first look at a simple implementation using Java's ArrayList and discuss two in-depth examples.

2.1 Implementing the Stack in Java

First, you should know that Java has a `Stack` class. Here, we're learning to build one ourselves. We'll represent the collection of items as an `ArrayList`, with the top of the stack being the tail of the list. Therefore, the push operation amounts to appending to the list, and the pop operation amounts to deleting the element at the tail. Both can be done efficiently on a list in $O(1)$ time. Hence, we can implement the Stack data type like so:

```
import java.util.*;

public class ArrayListStack<T> implements Stack<T> {
  private ArrayList<T> container;

  ArrayListStack() {
    container = new ArrayList<T>();
  }

  // add elt to the top of the stack
  public void push(T elt) { container.add(elt); }

  // remove and return the item at the top of the stack (most-recently added).
  public T pop() {
    T topElt = container.remove(container.size()-1);
    return topElt;
  }

  // return without removing the item at the top of the stack.
  public T top() { return container.get(container.size()-1); }

  // return a boolean indicating whether the stack is empty
  public boolean isEmpty() { return container.isEmpty(); }

}
```

2.2 Example I: Line Editor

In the old days, text editors weren't as convenient as the text editors you're used to today. The so-called line editors were pretty common. The basic idea is for you to type a sequence of letters that represent both the actual input text and commands. To avoid switching between modes, these key strokes interleave almost arbitrarily although special characters were designated as commands:

- The pound sign #, for example, has the effect of deleting the preceding character. Thus, the sequence of characters "`moo#d##eep`" results in the string "`meep`".

- Moreover, the at sign `@` has the effect of "killing" the line—that is, resetting the current line back to an empty line. This means, if we type `"hellomorbidworld@pretty planet"`, we'll simply get `"pretty planet"`.

With a stack, it is easy to implement a function that evaluates what the user types so we have the resulting string. Deleting the most recent character corresponds to popping the stack, and killing the line is basically starting a new stack. There's a slight problem because at the end, the elements in the stack are presented in the wrong order—the reverse of the original string. This, however, is easy to rectify: just reverse the string before we return. In code, we have the following:

```java
public class LineEditor {

  public static String computeLine(String inKeyStrokes) {
    Stack<Character> line = new ArrayListStack<>();

    for (int pos=0;pos<inKeyStrokes.length();pos++) {
      char thisChar = inKeyStrokes.charAt(pos);

      if (thisChar == '#')      line.pop();   // delete the most-recent stroke
      else if (thisChar == '@') line = new ArrayListStack<>(); // reset the line
      else                      line.push(thisChar); // here's the newest stroke
    }

    // At this point, line has the contents of the line - except in reversed
    // order. To fix this, we'll build a string and reverse it.
    StringBuilder netLineBuilder = new StringBuilder();
    while (!line.isEmpty()) { netLineBuilder.append(line.pop()); }
    return netLineBuilder.reverse().toString();
  }
}
```

2.3 Example II: Function Calls

We have studied recursive functions more or less as a black box. Internally, one magical thing that happens is the system/interpreter knows where we are. For example, in the factorial function, how does the system know to pass the return value of `fac(4)` to the body of `fac(5)`, which is waiting for it?

The answer is it keeps a stack. When a function is called, local variables (i.e. the environment) and where to return to are pushed onto the stack. And when a function returns, we pop the return address (where to go back to) and the environment of the place to go back to (to restore the local variables and such).

2.4 Example III: Expression Evaluator

How can we evaluate simple mathematical expressions such as $1 + 3/5.0 - 2 * 1.5$? With help from the stack, it is in fact not too difficult. To keep things simple, we'll deal with fully-parenthesized expressions. For example, $1 + 3/5.0 - 2 * 1.5$ would be (clumsily) written as `((1 + (3 / 5.0)) - (2 * 1.5))`. That is, every operator has two operands, and both that operator and its operands are surrounded by a pair of parentheses.

With this assumption, we can implement the two-stack algorithm of Dijkstra, which keeps two stacks—a value stack and an operator stack—and follows the following rules:

- Upon seeing a *value*, push that onto the value stack.
- Upon seeing an *operator*, push that onto the operator stack.
- Upon seeing a left parenthesis, ignore.
- Upon seeing a right parenthesis, pop the operator and two values, carry out the computation, and push the result back onto the value stack.

Let's work this example out by hand: `((1 + (3 / 5.0)) - (2 * 1.5))`.

Turning this idea into code isn't difficult:

```java
import java.util.*;

public class DijkstraExprEval {
  static double opr(double u, String op, double v) {
    if      (op.equals("+")) return u + v;
    else if (op.equals("-")) return u - v;
    else if (op.equals("*")) return u * v;
    else if (op.equals("/")) return u / v;
    else                     return Double.NaN;
  }

  public static double eval(List<String> tokens) {
    Stack<String> ops  = new ArrayListStack<>();
    Stack<Double> vals = new ArrayListStack<>();

    for (String tok : tokens) {
      if      (tok.equals("("))                ;
      else if (tok.equals("+")) ops.push(tok);
      else if (tok.equals("-")) ops.push(tok);
      else if (tok.equals("*")) ops.push(tok);
      else if (tok.equals("/")) ops.push(tok);
      else if (tok.equals(")")) {
        String op = ops.pop();
        double v  = vals.pop();
        double u  = vals.pop();
        vals.push(opr(u, op, v));
      }
      else vals.push(Double.parseDouble(tok)); // this is a number
    }

    return vals.top();
  }
}
```

Some Extra Convenient Functions: We can also write a function that takes fully-parenthesized expression in the form of a string and tokenizes that into a list form.

```java
import java.util.regex.*;
import java.util.*;

public class ExprEvalDriver {
  static List<String> tokenize(String expr) {
    String TOKEN_PATTERN = "(\\(|\\)|\\d+(\\.\\d+)*|[\\+\\-\\*/])";
    Pattern pattern = Pattern.compile(TOKEN_PATTERN);

    Matcher matcher= pattern.matcher(expr);
    List<String> tokens = new ArrayList<>();
    while (matcher.find()) { tokens.add(matcher.group()); }

    return tokens;
  }

  public static void main(String[] args) {
    List<String> tokens = tokenize(args[0]);
    System.out.println("ans: "+DijkstraExprEval.eval(tokens));
  }
}
```

Q: How do you implement a queue so that it runs in $O(1)$ all around?
Answer: You need a different kind of structure.

3 Linked Lists

We have been using the `ArrayList`, which is an implementation of the `List` interface. Let's now take a closer look at the `List` interface and start thinking about how we might support it ourselves. While Java's `List` interface has many more features, we'll strip it down to bare minimum. Our basic `List` interface supports the following:

- Can add an entry at the beginning (`prepend(e: T): void`)
- Can add an entry at the end (`add(e: T): void`)
- Can insert an entry anywhere (`insert(pos: int, e: T): void`)
- Can remove and return an entry (`remove(pos): T`)
- Can remove all entries (`clear(): void`)
- Can find out whether an element is part of the list (`contains(e:T):boolean`)
- Can retrieve the k-th element (`get(k: int): T`)
- Can tell the size of the list (`size():int`)
- Can see whether it's empty (`isEmpty():boolean`)

This leads to the following `ListInterface<T>`:

```
public interface ListInterface<T> {
  // Adds an entry e at the beginning of the list
  public void prepend(T e);

  // Adds an entry e at the end of the list
  public void add(T e);

  // Inserts an entry e into position pos. This means shifting all entries with
  // indices >= pos one forward. If pos is invalid, it will throw
  // IndexOfOutBoundsException.
  public void insert(int pos,T e);

  // Removes and returns the entry at position pos.
  public T remove(int pos);

  // Deletes all entries from the list
  public void clear();

  // Retrieves the element at position pos (starting from 0)
  public T get(int pos);

  // Returns whether the list contains a given element e.
  public boolean contains(T e);

  // Returns the number of entries in the list
  public int size();

  // Returns whether the list is empty
  public boolean isEmpty();
}
```

3.1 Chain of Linked Nodes

There are many ways to support the `LinkInterface` interface. But often, a chain of linked nodes—known as a *linked list*—is used as an alternative to an array. We'll save the discussion of their relative pros and cons to a later time. For now, let's come up with a plan on how to maintain a collection of items as a chain.

Perhaps, the most natural thing is to make each item an object on its own and chain them somehow, like so:

```
A --> B --> C --> D --> E
```

In code, this means we need a place to store the data item and a way to point to the next data item. The constructor is only there for convenience.

```
public class ListNode<T> {
  public T payload;
  public ListNode<T> next;

  ListNode(T payload_, ListNode<T> next_) {
    this.payload = payload_;
    this.next = next_;
  }
}
```

You can imagine that such a structure would support many of the operations listed above; however, there is crucial information that is still missing:

Where does the chain begin?

To fix this, we need something to point to the start of the chain. For easy access to the rear end of the list, we'll also keep a "pointer" to the node at the end of the chain. Pictorially:

```
A --> B --> C --> D --> E
^^                      ^^
hd                      tail
```

We'll also keep the size in our instance so we don't have to count every time the number of entries is requested. In sum, our class stores the following:

```
  ListNode<T> hd;      // head node of the list
  ListNode<T> tail;    // tail node of the list
  int sz;              // the number of entries
```

We'll quickly implement the easy functions and pay attention to the operations that require more care:

```
  // Deletes all entries from the list
  public void clear() { hd=null; tail=null; sz=0; }

  // Returns the number of entries in the list
  public int size() { return sz; }

  // Returns whether the list is empty
  public boolean isEmpty() { return sz==0; }
```