Stack

Introduction to Stack

- There are 2 commonly used data structures:
  - Stack (LIFO)
  - Queue (FIFO)
- A **stack** is a data structure that organize the stored data in a **Last In First Out (LIFO)** manner.
- To achieve the LIFO behavior, the stack only provide the following 2 methods to access the data stored in a stack:
  - `push(x)` : add `x` to the "top" of the stack.
  - `pop()` : remove the item at the "top" of the stack and return it.
  - The item removed by `pop()` is always the last item that was pushed.
- Method invocation/return:
  - If the order of method invocation is

    ```
    M1() --> M2() --> M3() --> M4()
    ```

  - Then the order in which the methods return form their invocation is the reverse order:

    ```
    M4() --> M3() --> M2() --> M1()
    ```

  - Some computer algorithms/processes with a natural LIFO behavior: undo algorithm in a text editor (it uses a stack to store the history of edit changes); back algorithm in a browser (it uses a stack to store the browser history)

The Stack Interface

- The stack interface definition:
  - The stack only defines a behavior on the access of the data stored in a stack: `pop()` must return the last item that was pushed
  - The stack does not specify how the data must be stored.
  - There are different ways to implement the same behavior

```java
public interface MyStackInterface<E> {
    boolean isEmpty(); // returns true if stack is empty
    boolean isFull(); // returns true if stack is full
    void push(E e); // pushes element e on the stack
    E pop(); // Remove the element at the top of the stack and return it
    E peek(); // Return the element at the top without removing it
}
```

Implementing the Stack with a fixed size array

- The basic implementation of a Stack is using:
    - A fixed size array to store the data items
    - A `stackTop` index variable to record the first open position in the array
- The initial state of the stack when it is instantiated (=created): `stackTop = 0` (can also use `stackTop = -1`)

```java
public class IntegerStack implements Stack<Integer> {
    private Integer[] item;
    private int stackTop;
    public IntegerStack(int N) { // Create a stack of size N
    item = new Integer[N];
    stackTop = 0;
    }
    public boolean isEmpty() { // Test if stack is empty
        return stackTop == 0;
    }
    public boolean isFull() { // Test if stack is empty
        return stackTop == item.length;
    }
    public void push(Integer e) {
        if (isFull()) {
            System.out.println("Full");
            return; // Or: throw an exception
        }
        item[stackTop] = e; // (1) store item
        stackTop++; // (2) increment stackTop
    }
    public Integer pop() {
        if (isEmpty()) {
            System.out.println("Empty");
            return null; // Or: throw an exception
        }
        stackTop--; // (1) decrement stackTop
        return item[stackTop]; // (2) return item
    }
}
```

-
    - See `IntegerStack.java` and `TestIntegerStack.java` .

Implement the stack with a dynamic array

- The stack can be implemented using a dynamic array

```java
public class IntegerStack implements Stack<Integer> {
    private Integer[] item;
    private int stackTop;
    private final double DELTA = 0.25;

    public IntegerStack(int N) { // Create a stack of size N
    item = new Integer[N];
    stackTop = 0;
    }
    public boolean isEmpty() { // Test if stack is empty
        return stackTop == 0;
    }
    public boolean isFull() { // Test if stack is empty
        return stackTop == item.length;
    }
    public void push(Integer e) {
        if (isFull()) {
            // Double the array size
            Integer[] temp = new int[2 * item.length];
            for (int i = 0; i < item.length; i++) {
                temp[i] = item[i];
            }
            item = temp;
        }
        item[stackTop] = e; // (1) store item
        stackTop++; // (2) increment stackTop
    }
    public Integer pop() {
        if (isEmpty()) {
            System.out.println("Empty");
            return null; // Or: throw an exception
        }
        stackTop--; // (1) decrement stackTop
        Integer retVal = item[stackTop];
        if (stackTop < DELTA * item.length && item.length >= 2) {
            // Reduce the array by half
            temp = new int[item.length / 2];
            for (int i = 0; i <= stackTop; i++) {
                temp[i] = item[i];
            }
            item = temp;
        }
        return retVal; // (2) return item
```

```
    }
}
```

- The value `DELTA` determines when we will reduce the size of the array: `DELTA` is a wastage threshold:
  - When only the fraction of `DELTA` of the array is being used, we will reduce the wastage.
  - Since we will reduce the array by half, `DELTA` must be at most 0.5. Otherwise, we will discard some valid entries in the stack.
  - `DELTA` = 0.25 is actually better than 0.5
- Running Time Analysis: Consider the `push()` algorithm using a dynamic array. On average, how many "store" statements are executed for each `push()` invocation?
  - When the stack is not full, the `push()` invocation will execute 1 store statement.
  - When the stack if full, the `push()` invocation will execute `(1 + item.length)` store statement.
  - Suppose we execute N `push()` operations:

# times exec `push()`	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	$\cdots$	N
# store statements to store item pushed	1	1	1	1	1	1	1	1	$\cdots$	1
# store statements to double array	1	2	0	4	0	0	0	8	$\cdots$	$M \leq N$

- Therefore, total store statements executed for N `push()` invocations:

$$(1 + 1 + \cdots + 1) + (1 + 2 + 4 + \cdots + M) \text{ where } M \leq N$$

  - Consider $S = 1 + 2 + 4 + \cdots + M$:

$$\begin{aligned} S &= 1 + 2 + 4 + \cdots + M \\ 2S &= \quad\;\; 2 + 4 + 8 + \cdots + 2M \\ S = 2S - S &= 2M - 1 \end{aligned}$$

  - Therefore, total store statements executed is

$$N + (2 * M - 1) \leq N + 2 * N - 1 = 3N - 1$$

  - Hence, average # store statement for `1 push()` invocation is $\dfrac{(3N-1)}{N} \approx 3.$

Generic Stack

- Java does not allow instantiation of a generic array, so the following code will cause error messages:

```
public class ArrayStack<T> implements Stack<T> {
    private T[] item;
    private int stackTop;

    public ArrayStack(int N) {
        item = new T[N]; // Create an array of T objects --> error
        stackTop = 0;
    }
    // other methods...
}
```

- However, there's a simple hack to work around this Java restriction.

```
public class ArrayStack<T> implements Stack<T> {
    private T[] item;
    private int stackTop;

    public ArrayStack(int N) {
        item = (T[]) new Object[N]; // Create an array of Object objects, and casting
        stackTop = 0;
    }
    // other methods...
}
```

- In this way, Java will report warning messages (not fatal errors), so our program will still compile and run.

```java
public class GenericStack<T> implements MyStackInterface<T> {
    private T[] item;
    private int stackTop;

    public GenericStack(int N) {
        item = (T[]) new Object[N]; // Create an array of Object objects
        // This will cause some warning, but it will compile (Java does not know if the
        // Why this will work: If we are working with unbounded generic types,
        // we know T will be interpreted as Object by Java. Then we will create
        // an array of Object, then cast it into our desired type T
        stackTop = 0;
    }
    @Override
    public boolean isEmpty() {
        return stackTop == 0;
    }

    @Override
    public boolean isFull() {
        return stackTop == item.length;
    }

    @Override
    public void push(T t) {
        // if the array is full, then double the size of the array
        if (isFull()) {
            System.out.println("Full");
            return;
        }
        item[stackTop] = t;
        stackTop++;
    }

    @Override
    public T pop() {
        if (isEmpty()) {
            System.out.println("Empty");
            return null; // or throw an exception
        }
        stackTop--; // (1) decrease stackTop
        return item[stackTop]; // return item
    }
```

```java
    @Override
    public T peek() {
        return item[stackTop - 1];
    }

    public String toString(){
        String result = "";
        for (int i = 0; i < stackTop; i++) {
            result += item[i] + " ";
        }
        return result;
    }
}
```

Java's Stack Library

- The Java library contains a generic `Stack` class: `java.util.Stack`
- To instantiate `Stack` objects:

```java
Stack<Integer> iStack = new Stack<>(); // Integer Stack
Stack<String> sStack = new Stack<>(); // String Stack
```

- The `Stack` class contains the following instance methods: `boolean empty()` ; `E peek()` ; `E push(E item)` ; `E pop()` .
- For some reasons, the `Stack` class is a subclass of the `Vector` class, which can access the sotred data using an index.
  - As a subclass, `Stack` inherits those methods:

  ```java
  get(int index); // Returns the element at the specified position
  remove(int index); // Removes the element at the specified position
  ```

  - However, this inheritance makes the FIFO behavior not guaranteed.

Application of Stack: Reverse Polish Expression Evaluation

- There are 3 ways to write arithmetic expressions:
  - In-fix: operators are placed between their operands: $(A+B)\times C = (A+B)\times C$.
  - Pre-fix: operators are placed before their operands: $\times + A\ B\ C\ = (A+B)\times C$.

- Post-fix: operators are placed after their operands: $A\ B + C\times = (A + B) \times C$.
- The pre-fix and post-fix notations do no use parenthesis to write arithmetic expressions.

- Reverse Polish Notation (RPN):
  - The operator always follows its (2) operands: `3 4 + ==> 3 + 4 = 7`
  - When we evaluate an operation in RPN, the result is used an operand of another operation: `3 4 + 1 - ==> 7 1 - ==> 6`
  - Conclusion:
    - Each operator will operate on its proceeding 2 operands.
    - Each operator will produce a result that will be the operand of some subsequent operator.
  - We use a stack to store the operands. Whenever we reach an operator, we evaluate the operation with the two operands at the top of the stack.

```java
import java.util.Stack;
public class EavluatePRN {
    public static void main(String[] args) {
        System.out.println(evalRPN(args));
    }

    /**
     * Reverse Polish Notation (RPN):
     *      3 4 + ===> 3 + 4 = 7
     *      - Each operator will operate on its proceeding 2 operands
     *      - Each operator will produce a result that will be the operand of some subs
     * We will use a Stack to implement this algorithm
     * @param inp = array of String representing an RPN expression (e.g.: "3" "4" "+')
     */
    public static int evalRPN(String[] inp) {
        Stack<Integer> opStack = new Stack<>(); // Stack containing the prior oprands
        String s; // Help variable containg the next symbol
        for (int i = 0; i < inp.length; i++) {
            s = inp[i]; // s = next item/symbol in input (as String !)
            if (s.equals("+") || s.equals("-") || s.equals("x") || s.equals("/")) {
                // the next symbol is an operator
                int o2 = opStack.pop(); // Get the last 2 operands
                int o1 = opStack.pop();

                int r = operate(s, o1, o2); // Perform operation
                opStack.push(r); // Save result (operand) on stack

            } else { // the next symbol is an oprands
                opStack.push(Integer.parseInt(s)); // Save number as Integer
            }
        }
        return opStack.pop(); // Return result (was saved on stack)
    }

    public static int operate(String op, int o1, int o2) {
        if (op.equals("x")) { // Multiply
            return o1 * o2;
        } else if (op.equals("/")) {
            return o1/o2;
        } else if (op.equals("+")) {
            return o1 + o2;
        }
        else if (op.equals("-")) {
```

```
            return o1 - o2;
        } else {
            return 0;
        }
    }
}
```

Stack Application: Edsger Dijkstra Algorithm for Fully Parenthesized Arithmetic Expression

- Problem description:
    - We are given a fully parenthesized arithmetic expression using only `x`, `/`, `+`, and `-` operations.
    - Write an algorithm to evaluate expressions in this form.
    - We will need 2 stakcs:
        - An `operand` stack that stores the operands in the input, and
        - An `operator` stack that stores the operators in the input.
- Algorithm:
    - Find the first occurrence of a right parenthesis `)`: observe the last 2 operands and the last operation prior to the right parenthesis.
        - This guarantees the most inner `()` will be the first to be evaluated.
        - The result of an operation must be pushed on to operand stack.
    - Then, we can reduce the parenthesis and find the next earliest occurrence of the right parenthesis.
    - Repeat the steps until the input array is exhausted.
    - The left parenthesis `(` does not convey and information.

```
import java.util.Stack;
public class Dijkstra2Stackalg {
    public static Integer eval(String[] inp) {
        Stack<Integer> operandStck = new Stack<>();
        Stack<String> operatorStck = new Stack<>();
        String s;

        for (int i = 0; i < inp.lengthl i++) {
            s = inp[i];
            if (s.equals("(")) {
                // do nothing
            } else if (s.equals("+") || s.equals("-") || s.equals("x") || s.equals("/")
                operatorStck.push(s);
            } else if (s.equals(")")) { // compute the must inner ()
                int o2 = operandStck.pop();
                int o1 = operandStck.pop();
                String op = operatorStck.pop();
                int r = operate(op, o1, o2);
                operandStck.psuh(r);
            } else { // s is a number
                operandStck.push(s);
            }
        }
        return operandStck.pop();
    }
}
```