```java
import java.util.*;

public class FreeListAllocator {
    private static class Block {
        int start, length;
        Block(int s, int l) { start = s; length = l; }
    }

    private final LinkedList<Block> freeList;

    public FreeListAllocator(int heapSize) {
        freeList = new LinkedList<>();
        freeList.add(new Block(0, heapSize)); // Initially entire heap is
free
    }

    public int malloc(int size) {
        for (ListIterator<Block> it = freeList.listIterator();
it.hasNext();) {
            Block block = it.next();
            if (block.length >= size) {
                int allocStart = block.start;
                if (block.length == size) {
                    it.remove();
                } else {
                    block.start += size;
                    block.length -= size;
                }
                return allocStart;
            }
        }
        return -1; // No fitting block found
    }

    public void free(int ptr, int size) {
        ListIterator<Block> it = freeList.listIterator();
        while (it.hasNext()) {
            if (ptr < it.next().start) {
                it.previous();
                break;
```

```
            }
        }
        Block newBlock = new Block(ptr, size);
        it.add(newBlock);

        // Coalesce with previous if adjacent
        int prevIndex = it.previousIndex() - 1;
        if (prevIndex >= 0) {
            Block prev = freeList.get(prevIndex);
            if (prev.start + prev.length == newBlock.start) {
                prev.length += newBlock.length;
                freeList.remove(newBlock); // remove by reference, not
iterator remove
                newBlock = prev;
            }
        }

        // Coalesce with next if adjacent
        if (it.hasNext()) {
            Block next = it.next();
            if (newBlock.start + newBlock.length == next.start) {
                newBlock.length += next.length;
                freeList.remove(next); // remove by reference, avoid
iterator remove
            }
        }
    }

    public List<int[]> freeList() {
        List<int[]> res = new ArrayList<>();
        for (Block b : freeList) {
            res.add(new int[]{b.start, b.length});
        }
        return res;
    }

    public String freeBlocksView() {
        StringBuilder sb = new StringBuilder();
        for (Block b : freeList)
sb.append("[").append(b.start).append(",").append(b.length).append("] ");
```

```
        return sb.toString().trim();
    }

    public static void main(String[] args) {
        FreeListAllocator allocator = new FreeListAllocator(20);
        System.out.println("free: " + allocator.freeBlocksView());
        int a = allocator.malloc(8);
        System.out.println("malloc(8) → " + a + " // free: " +
allocator.freeBlocksView());
        int b = allocator.malloc(4);
        System.out.println("malloc(4) → " + b + " // free: " +
allocator.freeBlocksView());
        allocator.free(a, 8);
        System.out.println("free(0,8) // free: " +
allocator.freeBlocksView());
        int c = allocator.malloc(6);
        System.out.println("malloc(6) → " + c + " // free: " +
allocator.freeBlocksView());
        allocator.free(b, 4);
        System.out.println("free(8,4) // free: " +
allocator.freeBlocksView());
        allocator.free(12, 8);
        System.out.println("free(12,8) // free: " +
allocator.freeBlocksView());
        System.out.println("Final free: " + allocator.freeBlocksView());
    }
}
```

```
PS E:\JAVA PROGRAMS\steparyansingh\year2\dsa\lab> javac *.java
PS E:\JAVA PROGRAMS\steparyansingh\year2\dsa\lab> java FreeListAllocator
free: [0,20]
malloc(8) ? 0 // free: [8,12]
malloc(4) ? 8 // free: [12,8]
free(0,8) // free: [0,8] [12,8]
malloc(6) ? 0 // free: [6,2] [12,8]
free(8,4) // free: [6,6] [12,8]
free(12,8) // free: [6,6] [12,8] [12,8]
Final free: [6,6] [12,8] [12,8]
```

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

public class InfixExpressionEvaluator {
    private static final Set<String> FUNCTIONS = Set.of("min", "max",
"abs");
    private static final Set<String> OPERATORS = Set.of("+", "-", "*",
"/", "%", "^");
    private static final Map<String, Integer> PRECEDENCE = Map.of(
            "+", 1, "-", 1,
            "*", 2, "/", 2, "%", 2,
            "^", 3,
            "u-", 4  // unary minus has highest precedence
    );
    private static final Set<String> RIGHT_ASSOC = Set.of("^", "u-");

    private final Map<String, Integer> env;

    public InfixExpressionEvaluator(Map<String, Integer> env) {
        this.env = env;
    }

    private boolean isOperator(String token) {
        return OPERATORS.contains(token) || token.equals("u-");
    }

    private boolean isFunction(String token) {
        return FUNCTIONS.contains(token);
    }

    private int getPrecedence(String op) {
        return PRECEDENCE.getOrDefault(op, -1);
    }

    private boolean isRightAssociative(String op) {
        return RIGHT_ASSOC.contains(op);
    }

    // Tokenize input expression into list of tokens
    private List<String> tokenize(String expr) throws Exception {
```

```
        List<String> tokens = new ArrayList<>();
        Pattern pattern = Pattern.compile("\\d+|[a-zA-Z_]+|\\S");
        Matcher matcher = pattern.matcher(expr);
        while (matcher.find()) {
            tokens.add(matcher.group());
        }
        return tokens;
    }

    // Convert infix tokens to RPN using Shunting-Yard algorithm
    private List<String> toRPN(List<String> tokens) throws Exception {
        List<String> output = new ArrayList<>();
        Deque<String> stack = new ArrayDeque<>();

        String prevToken = null;

        for (String token : tokens) {
            if (token.matches("\\d+")) {
                output.add(token);
                prevToken = "operand";
            } else if (token.matches("[a-zA-Z_]+")) {
                if (isFunction(token)) {
                    stack.push(token);
                    prevToken = "func";
                } else {
                    // Variable
                    output.add(token);
                    prevToken = "operand";
                }
            } else if (token.equals(",")) {
                while (!stack.isEmpty() && !stack.peek().equals("(")) {
                    output.add(stack.pop());
                }
                if (stack.isEmpty() || !stack.peek().equals("(")) {
                    throw new Exception("ERROR");
                }
                prevToken = "comma";
            } else if (token.equals("(")) {
                stack.push(token);
                prevToken = "left_paren";
```

```
            } else if (token.equals(")")) {
                while (!stack.isEmpty() && !stack.peek().equals("(")) {
                    output.add(stack.pop());
                }
                if (stack.isEmpty()) {
                    throw new Exception("ERROR");
                }
                stack.pop(); // remove '('
                if (!stack.isEmpty() && isFunction(stack.peek())) {
                    output.add(stack.pop());
                }
                prevToken = "right_paren";
            } else if (isOperator(token)) {
                // unary minus handling
                if (token.equals("-") && (prevToken == null ||
prevToken.equals("operator") ||
                        prevToken.equals("left_paren") ||
prevToken.equals("comma"))) {
                    token = "u-";
                }
                while (!stack.isEmpty() && isOperator(stack.peek())) {
                    String top = stack.peek();
                    int topPrec = getPrecedence(top);
                    int tokenPrec = getPrecedence(token);

                    if ((topPrec > tokenPrec) ||
                            (topPrec == tokenPrec &&
!isRightAssociative(token))) {
                        output.add(stack.pop());
                    } else {
                        break;
                    }
                }
                stack.push(token);
                prevToken = "operator";
            } else {
                throw new Exception("ERROR");
            }
        }
```

```
        while (!stack.isEmpty()) {
            String top = stack.pop();
            if (top.equals("(") || top.equals(")")) {
                throw new Exception("ERROR");
            }
            output.add(top);
        }
        return output;
    }

    // Evaluate RPN expression
    private int evalRPN(List<String> rpn) throws Exception {
        Deque<Integer> stack = new ArrayDeque<>();

        for (String token : rpn) {
            if (token.matches("\\d+")) {
                stack.push(Integer.parseInt(token));
            } else if (env.containsKey(token)) {
                stack.push(env.get(token));
            } else if (isOperator(token)) {
                if (token.equals("u-")) {
                    if (stack.isEmpty()) throw new Exception("ERROR");
                    int val = stack.pop();
                    stack.push(-val);
                } else {
                    if (stack.size() < 2) throw new Exception("ERROR");
                    int b = stack.pop();
                    int a = stack.pop();
                    stack.push(applyOp(token, a, b));
                }
            } else if (isFunction(token)) {
                if (token.equals("abs")) {
                    if (stack.isEmpty()) throw new Exception("ERROR");
                    int val = stack.pop();
                    stack.push(Math.abs(val));
                } else if (token.equals("min") || token.equals("max")) {
                    if (stack.size() < 2) throw new Exception("ERROR");
                    int b = stack.pop();
                    int a = stack.pop();
```

```
                    stack.push(token.equals("min") ? Math.min(a, b) :
Math.max(a, b));
                } else {
                    throw new Exception("ERROR");
                }
            } else {
                throw new Exception("ERROR");
            }
        }

        if (stack.size() != 1) throw new Exception("ERROR");
        return stack.pop();
    }

    private int applyOp(String op, int a, int b) throws Exception {
        switch (op) {
            case "+": return a + b;
            case "-": return a - b;
            case "*": return a * b;
            case "/":
                if (b == 0) throw new Exception("ERROR");
                return a / b; // integer division truncates toward zero
            case "%":
                if (b == 0) throw new Exception("ERROR");
                return a % b;
            case "^":
                if (b < 0) throw new Exception("ERROR");
                return (int)Math.pow(a, b);
        }
        throw new Exception("ERROR");
    }

    public String evaluate(String expr) {
        try {
            List<String> tokens = tokenize(expr);
            List<String> rpn = toRPN(tokens);
            int result = evalRPN(rpn);
            return Integer.toString(result);
        } catch (Exception e) {
            return "ERROR";
```

```
        }
    }

    // For demonstration with your examples
    public static void main(String[] args) {
        Map<String, Integer> env1 = new HashMap<>(); // empty
        Map<String, Integer> env3 = Map.of("x", -2, "y", -7);
        Map<String, Integer> env4 = Map.of("a", 1);

        InfixExpressionEvaluator evaluator1 = new
InfixExpressionEvaluator(env1);
        InfixExpressionEvaluator evaluator3 = new
InfixExpressionEvaluator(env3);
        InfixExpressionEvaluator evaluator4 = new
InfixExpressionEvaluator(env4);

        System.out.println(evaluator1.evaluate("3 + 4 * 2 / (1 - 5) ^
2^3"));           // Output: 3
        System.out.println(evaluator1.evaluate("min(10, max(2, 3*4))"));
// Output: 10
        System.out.println(evaluator3.evaluate("-(x) + abs(y)"));
// Output: 9
        System.out.println(evaluator4.evaluate("a + b"));
// Output: ERROR
    }
}
```

```
PS E:\JAVA PROGRAMS\steparyansingh\year2\dsa\lab> java InfixExpressionEvaluator
3
10
9
ERROR
```