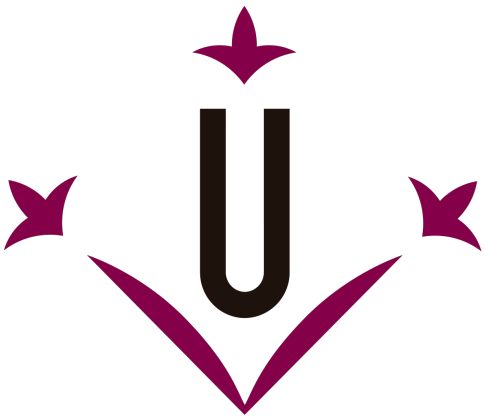

Universitat de Lleida

Stacks and Recursive Iterative Transformation

Data Structure

Escola Politècnica Superior

Computer Science

Yhislaine Nataly, Jaya Salazar

Valeria Adriana, Escalera Flores

01.11.2024

Contents

1 Introduction

This report presents the results of Lab 2, where concepts of transformation from recursive to iterative methods have been studied through the use of the stack data structure. The ArrayStack[1] class has been implemented to manage these structures and, subsequently, the methodology for converting recursive functions, such as the calculation of Catalan numbers, to an iterative format has been applied.

2 Array Stack

The first part of the practice consisted of the implementation of the `ArrayStack` class, which is based on the use of an array list to operate with the elements of the stack. The implementation of `ArrayStack` conforms to the `Stack <E>` interface, which defines the basic methods for manipulating a stack: `ArrayStack()`, `push(E elem)`, `top()`, `pop()` and `isEmpty()`.

2.1 Implementation

2.1.1 `ArrayStack()`

On the one hand, the `ArrayStack()` [3] constructor initialises the stack by creating an array, `stackArray`, which is used to store the stack elements and has a generic type `E`. As in Java it is not possible to create array instances directly with generic types, a cast is made so that stackArray can handle elements of the specified type. On the other hand, to suppress the Java warning generated by such a cast, `@SuppressWarnings('unchecked')` is used, indicating that this detail does not affect the safe operation of the implementation.

In this context, the prompt specified an initial size of 10. Initially, we established this value to 10 during the creation process. However, we opted to implement the methodology discussed in class, which involves defining the default size as a `static final` variable.

```
01 |     private static final int DEFAULT_SIZE = 10;
```

```
01 |    @SuppressWarnings("unchecked")
```

```
02 |     public ArrayStack() {
03 |         this.stackArray = (E[]) new Object[DEFAULT_SIZE];
04 |         this.top = 0;
05 |     }
```

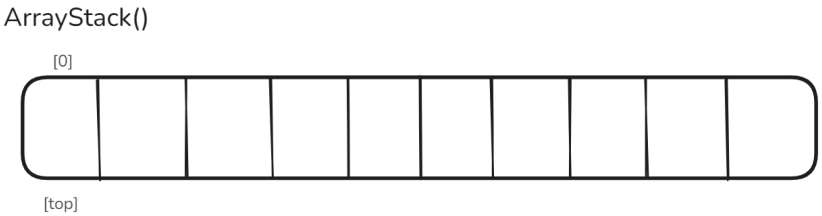

Figure 1: Diagram illustrating the outcome of the constructor `ArrayStack()`

2.1.2 `push(E elem)`

In this `push(E elem)` method, an elem element is added to the stack. First, the method checks if the stack has reached its maximum capacity and makes sure there is room for the new element. Then, elem is placed at the position indicated by top, and top is incremented to point to the next free position on the stack.

```
01 |     @Override
02 |     public void push(E elem) {
03 |         if(this.top == stackArray.length) resize();
04 |         this.stackArray[top] = elem;
05 |         this.top++;
06 |     }
```

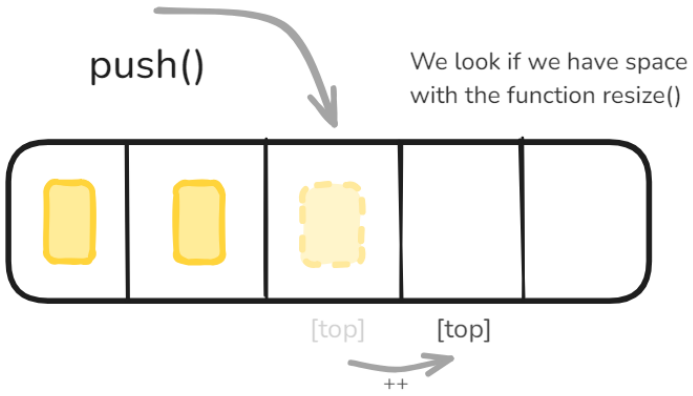

Figure 2: Diagram illustrating the process of the `push()`

2.1.3 `resize()`

The `resize()` method is responsible for doubling the capacity of the stackArray array when it is full, thus allowing the storage of more elements. To do this, it creates a new `auxStackArray` array twice the length of the `stackArray`. Also, it is necessary to cast `Object[ ]` to `E[ ]`, and `@SuppressWarnings('unchecked')` is used again to avoid compilation warnings about the cast. The method copies each element of the original array to the new expanded array and, at the end, `stackArray` is replaced by `auxStackArray`, effectively increasing its capacity without losing existing data. This ensures that the stack is dynamically expandable when more space is needed.

```
01 | @SuppressWarnings("unchecked")
02 |    private void resize (){
03 |        E[] auxStackArray = (E[]) new Object[this.stackArray.length *
       2];
04 |        for(int i = 0; i <stackArray.length; i++){
05 |            auxStackArray[i] = stackArray[i];
06 |        }
07 |        stackArray = auxStackArray;
08 |    }
```

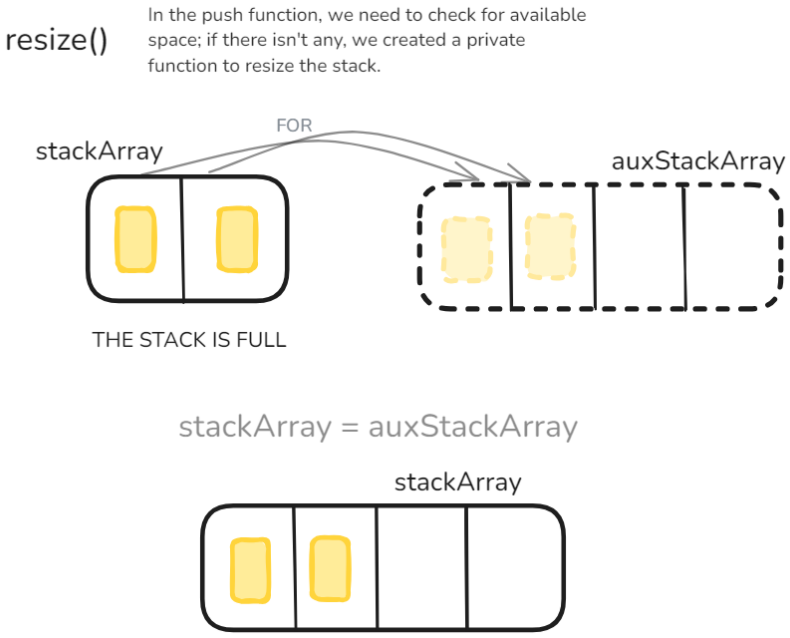

Figure 3: The diagram shows the concept of the `resize()`

2.1.4 `top()`

This method allows consulting the element at the top of the stack safely and without altering its contents. This means that it returns the last element added to the stack without removing it. First, it checks if the stack is empty by calling `isEmpty()`, and if it is, it throws a `NoSuchElementException` to indicate that there are no elements available to return. If the stack is not empty, `top()` returns the element located at the `top - 1` position, which represents the top of the stack.

```
01 |    @Override
02 |     public E top() {
03 |         if(isEmpty()) throw new NoSuchElementException();
04 |         return this.stackArray[top-1];
05 |     }
```

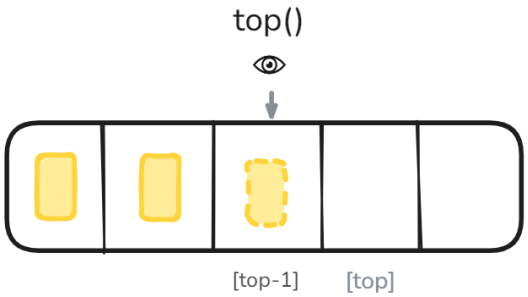

Figure 4: The diagram shows the main idea of `top()`

2.1.5 `pop()`

The `pop()` method removes the last element added to the stack. First, it checks if the stack is empty using `isEmpty()`. If it is, it throws a `NoSuchElementException`. If the stack contains elements, `pop()` decrements the top index by 1, 'moving' the top of the stack to the previous position. It then sets the value of `stackArray[top]` to `null` to free up the space.

```
01 |     @Override
02 |     public void pop() {
03 |         if(isEmpty()) throw new NoSuchElementException();
```

```
        this.top--;
        this.stackArray[top] = null;
    }
```

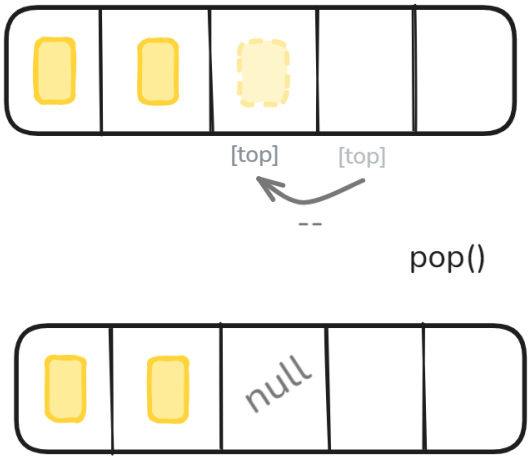

Figure 5: The diagram illustrates the process of the `pop()`

2.1.6 `isEmpty()`

The `isEmpty()` method checks if the stack is empty. If top is 0, the method returns true, indicating that the stack is empty; otherwise, it returns false. This method is a quick and direct check of the stack status, useful to avoid invalid operations, such as `pop()` or `top()`, when there are no elements.

```
    @Override
    public boolean isEmpty() {
        return this.top == 0;
    }
```

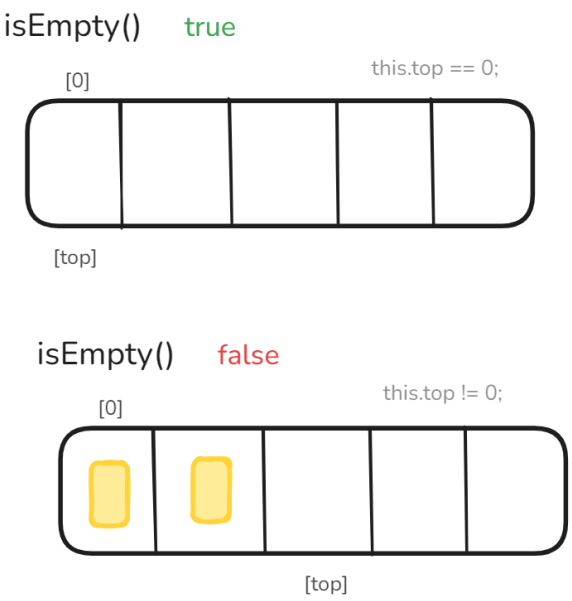

Figure 6: The diagram shows how `isEmpty()` works

2.2 Tests

This test suite verifies the functionality of our `ArrayStack` implementation using JUnit. These tests cover essential scenarios, checking both expected performance and exception handling. Each test follows the next code structure:

```
01 | class ArrayStackTest {
02 |
03 |     @Test
04 |     @DisplayName("Brief and specific test description")
05 |     void testMethodName() {
06 |         // Initial setup (Arrange)
07 |         var stack = new ArrayStack<Integer>();
08 |
09 |         // Operations (Act)
10 |         // Calls to ArrayStack methods, for example:
11 |         // stack.push(1);
12 |
13 |         // Verifications (Assert)
14 |         // Use assertEquals, assertThrows, assertTrue, etc.
15 |     }
16 | }
```

The test methods use `snake_case` naming for improved readability and cover essential behaviors, such as exception handling (for example, attempting to `pop()` from an empty stack), maintaining `LIFO` (*Last In, First Out*) order, and managing capacity limits. Each test includes a `@DisplayName` annotation for quick contextual understanding, which aids in interpreting test results.
Overall, this suite ensures that the `ArrayStack` performs reliably under both standard operations and edge cases, with tests that are clear and well-organized.

2.2.1 `push_adds_elements_to_stack()`

In this test, the focus is on the `push()` method, which adds an element to the top of the stack. The test initializes a new stack called `arrayStack` and pushes the integer value 10 onto it. After performing the push operation, it uses `assertEquals` to verify that the top element of the stack is now 10. This confirms that the push operation has successfully added the element to the stack, and that the stack's state reflects this change accurately.

```
01 |     @Test
02 |     @DisplayName("push adds elements to the stack")
03 |     void push_adds_element_to_stack() {
04 |         var arrayStack = new ArrayStack<Integer>();
05 |         arrayStack.push(10);
06 |         assertEquals(10, arrayStack.top());
07 |     }
```

2.2.2 `pop_removes_top_element()`

This test checks if the pop() method correctly removes and returns the top element from the stack. It begins by creating a new instance of `ArrayStack<Integer>` called stack, then pushes two integers: 5 and 10. After these operations, it calls `stack.pop()`, which should remove the top element (10). The test then asserts that the new top element is 5 using `assertEquals`. This confirms that the pop operation works as intended, effectively removing the last pushed element and maintaining the correct order of elements in the stack.

```
    @Test
    @DisplayName("pop removes the top element from the stack")
    void pop_removes_top_element() {
        var stack = new ArrayStack<Integer>();
        stack.push(5);
        stack.push(10);
        stack.pop();
        assertEquals(5, stack.top());
    }
```

2.2.3 pop_on_empty_stack_throws_exception()

Similar to the first test, this one again verifies that popping from an empty stack raises a `NoSuchElementException`. It creates a new instance of `ArrayStack<Integer>` named stack and immediately calls `assertThrows`, passing in a reference to `stack.pop()`. If no exception is thrown, or if a different exception is thrown, this test will fail. This reinforces proper error handling in stack operations when attempting to remove elements from an empty structure.

```
    @Test
    @DisplayName("pop on empty stack throws NoSuchElementException")
    void pop_on_empty_stack_throws_exception() {
        var stack = new ArrayStack<Integer>();
        assertThrows(NoSuchElementException.class, stack::pop);
    }
```

2.2.4 top_returns_top_element()

This test evaluates whether the `top()` method correctly retrieves the top element without modifying the stack's state. A new stack instance named stack is created, and two integers (20 and 30) are pushed onto it in that order. The test first checks that calling stack.top() returns 30, which confirms that it accurately reflects the last pushed value. It then calls top() again and asserts that it still returns 30, demonstrating that calling this method does not alter the stack's contents.

```
    @Test
    @DisplayName("top returns the top element without removing it")
```

```
03 |     void top_returns_top_element() {
04 |         var stack = new ArrayStack<Integer>();
05 |         stack.push(20);
06 |         stack.push(30);
07 |         assertEquals(30, stack.top());
08 |         assertEquals(30, stack.top());
09 |     }
```

2.2.5 `top_on_empty_stack_throws_exception()`

This test checks if calling `top()` on an empty stack results in a proper exception being thrown. A new instance of `ArrayStack<Integer>` called stack is created but remains empty. The test uses `assertThrows` to ensure that invoking `stack.top()` raises a `NoSuchElementException`. This verifies that error handling is consistent across different operations when dealing with an empty stack.

```
01 |     @Test
02 |     @DisplayName("top on empty stack throws NoSuchElementException")
03 |     void top_on_empty_stack_throws_exception() {
04 |         var stack = new ArrayStack<Integer>();
05 |         assertThrows(NoSuchElementException.class, stack::top);
06 |     }
```

2.2.6 `resize_doubles_capacity()`

In this test, we assess whether the stack can dynamically resize when its capacity is exceeded. A new instance of `ArrayStack<Integer>` named stack is created, and a loop pushes 15 integers onto it (assuming its initial capacity is less than 15). After all push operations are completed, an assertion checks if the top element equals 14 (the last pushed value). This indicates that all elements were successfully added and that resizing occurred as expected when reaching capacity limits.

```
01 |     @Test
02 |     @DisplayName("resize doubles the capacity of the stack when full")
03 |     void resize_doubles_capacity() {
04 |         var stack = new ArrayStack<Integer>();
05 |         for (int i = 0; i < 15; i++) {
06 |             stack.push(i);
```

```
        }
        assertEquals(15, stack.top() + 1);
    }
```

2.2.7 isEmpty_checks_if_stack_is_empty()

This test examines how well the method `isEmpty()` functions under various conditions. It starts with a new instance of `ArrayStack<Integer>`, checking initially if it returns true (indicating it's empty). Next, it pushes one integer (1) onto the stack and asserts that calling `isEmpty()` now returns false, confirming that there is at least one element present. Following this, it pops the single element off and checks again if it returns true, validating that after removal, the stack is indeed empty once more.

```
    @Test
    @DisplayName("isEmpty returns true if the stack is empty, false
   otherwise")
    void isEmpty_checks_if_stack_is_empty() {
        var stack = new ArrayStack<Integer>();
        assertTrue(stack.isEmpty());

        stack.push(1);
        assertFalse(stack.isEmpty());
        stack.pop();
        assertTrue(stack.isEmpty());
    }
```

2.2.8 push_elements()

This test focuses on pushing multiple elements onto the stack and verifying their order upon popping them off. The test creates a new instance of `ArrayStack<Integer>`, pushing three integers sequentially: 1, 2, and 3. After pushing these values, it asserts that calling `top()` returns 3 (the last pushed value). Then, it pops off this top value and checks if the next top value is now 2, followed by another pop to confirm that 1 remains as the last item on the stack before it becomes empty. This ensures that elements are stored in `LIFO` (*Last In First Out*) order.

```
    @Test
    @DisplayName("push multiple elements and should be in order")
```

```
03 |     void push_elements() {
04 |         var arrayStack = new ArrayStack<Integer>();
05 |         arrayStack.push(1);
06 |         arrayStack.push(2);
07 |         arrayStack.push(3);
08 |         assertEquals(3, arrayStack.top());
09 |         arrayStack.pop();
10 |         assertEquals(2, arrayStack.top());
11 |         arrayStack.pop();
12 |         assertEquals(1, arrayStack.top());
13 |     }
```

2.2.9 `push_pop_elements()`

This test evaluates a mixed sequence of push and pop operations to ensure consistency in behavior across these actions. It initializes a new instance of `ArrayStack<Integer>` and pushes two integers: first pushing 1 followed by 2. After pushing these values, it pops one integer off (removing 2) and then pushes another integer (3). The test checks if calling `top()` returns 3 after this sequence before popping again to verify that 1 becomes the new top element after removing 3. This demonstrates how well push/pop operations can interleave while maintaining correct state.

```
01 |     @Test
02 |     @DisplayName("Sequence of push and pop")
03 |     void push_pop_elements() {
04 |         var arrayStack = new ArrayStack<Integer>();
05 |         arrayStack.push(1);
06 |         arrayStack.push(2);
07 |         arrayStack.pop();
08 |         arrayStack.push(3);
09 |         assertEquals(3, arrayStack.top());
10 |         arrayStack.pop();
11 |         assertEquals(1, arrayStack.top());
12 |     }
```

2.2.10 `check_push_pop()`

This comprehensive test performs multiple cycles of push and pop operations to stress-test stack behavior under repeated use cases. A loop iterates five times; during each iteration, two values are pushed: one being an index (i) and another being ten times that index (i * 10). After each push operation, assertions check if the top value matches what was just pushed (i * 10). Following this verification, each value is popped off in reverse order while also checking if popping leaves behind only what was previously pushed (i). Finally, after popping both values in each cycle, it asserts whether or not the stack is empty at each step. This thorough testing helps ensure robustness against boundary conditions.

```
01 |     @Test
02 |     @DisplayName("multiple Push Pop cycle")
03 |     void check_push_pop() {
04 |         var arrayStack = new ArrayStack<Integer>();
05 |         for (int i = 0; i < 5; i++) {
06 |             arrayStack.push(i);
07 |             arrayStack.push(i * 10);
08 |             assertEquals(i * 10, arrayStack.top());
09 |             arrayStack.pop();
10 |             assertEquals(i, arrayStack.top());
11 |             arrayStack.pop();
12 |             assertTrue(arrayStack.isEmpty());
13 |         }
14 |     }
```

In summary, these tests cover the main use cases and error scenarios for the `ArrayStack` class. Through them, we verify that the implementation complies with the specifications of a dynamic stack, correctly handling both automatic array growth and exceptions when trying to operate on an empty stack. This ensures that the class is safe and suitable for use in scenarios requiring a stack-like data structure.

3 Transformation to Iterative

In the second exercise of our practice session, we aimed to transform iterative functions into their recursive counterparts. For this task, we focused on two well-known examples: the Fibonacci[2] sequence and the Factorial. Our methodology relied on utilizing the operations associated with a stack, which we had previously implemented in an earlier exercise.

Throughout this process, we delved deeply into understanding the intricacies of how to transition from a sequence to its recursive representation, and conversely, how to revert from recursive forms back to iterative ones. This exploration included examining various programming constructs and techniques, such as switch statements, case statements, and stack functions, allowing us to gain a comprehensive understanding of the different methodologies involved in these conversions. This hands-on approach not only clarified the theoretical concepts but also enhanced our practical programming skills.

3.1 Catalan Number

In order to implement the function effectively, it was essential to first comprehend the mechanism behind Catalan numbers. The PDF provided in the practice session offered valuable insights, enabling us to manually work through an example of the calculations.[1]

$$C_2 = C_0 * C_1 + C_1 * C_0 \tag{1}$$

$$C_1 = C_0 * C_0 \tag{2}$$

$$C_0 = 1 \tag{3}$$

$$C_1 = 1 \Rightarrow C_2 = 2 \tag{4}$$

TTo effectively understand the calculation of Catalan numbers, it is crucial to identify the specific variables required for an iterative implementation. Each variable plays a distinct role in mimicking the recursion typically found in such calculations.

[1]Specifically, we will illustrate the process of deriving the second Catalan number, C_2. This example will serve to elucidate the recursive nature of the Catalan number when n equals 2.

n: This variable signifies the specific number for which we aim to compute the corresponding Catalan number. Essentially, it serves as the input parameter of our calculations.

catalan1 and **catalan2**: These two variables are vital for storing intermediate results derived from recursive calls. Specifically, `catalan1` holds the result of the recursive calculation for the current index, expressed as `catalanRecWhile(i)`, while `catalan2` captures the result for the complementary index, represented by `catalanRecWhile(n - i - 1)`. By retaining these partial results, we can build towards the final output without depending on the recursive call stack.

entryPoint: This variable keeps track of our current location within the iterative process, acting as a marker to delineate which phase of the calculation we are in. It can signify whether we are at the beginning of the process, after computing `catalan1`, or after obtaining `catalan2`. This organization ensures smooth transitions through the various stages of the calculation.

i: Serving as a loop counter, this variable facilitates the simulation of multiple recursive calls by iterating through different values. As we progress through the loop, `i` enables us to explore all necessary recursive paths that contribute to the final result.

These variables collectively enable us to replicate the behavior of recursion in an iterative framework. By maintaining the current calculation state and storing intermediate results, we circumvent the need for a traditional recursive approach. In doing so, we can also establish the structure of the `entryPoint`, which consists of a single `CALL` and two designated `RESUME` points, allowing for an organized execution flow throughout the calculation process.

3.2 Code

Initially, we introduce the `EntryPoint` class. [2]. This class is tasked with defining the current state of the context, which will enable us to ascertain the appropriate handling of each respective situation.

```
01 |    private enum EntryPoint {
02 |         CALL, RESUME1, RESUME2
03 |     }
```

In the following section, we will discuss the private static `Context` class, wherein we will declare the required variables. The first variable is `n`, which will function as the input parameter. Subsequently, we will define the variables `catalan1`, `catalan2`, `i`, and `entryPoint`, each assigned its respective value. The integer variables will be initialized to 0, while `entryPoint` will be assigned the value of `CALL`.

```
01 | private static class Context {
02 |         final int n;
03 |         long catalan1;
04 |         long catalan2;
05 |         int i;
06 |
07 |         EntryPoint entryPoint;
08 |
09 |         Context(int n){
10 |             this.n = n;
11 |             this.catalan1 = 0L;
12 |             this.catalan2 = 0L;
13 |             this.entryPoint = EntryPoint.CALL;
14 |             this.i = 0;
15 |         }
16 |
17 |     }
```

In the `catalanIter` function, we will convert the implementation to an iterative approach that utilizes stack-based actions. It is imperative to declare the `stack` variable, which will be responsible for storing the `context` attributes, and to perform an initial push with the

[2] In the accompanying diagram the term "Call" is abbreviated as "C," while "RESUME1" is represented as "R1" and "RESUME2" as "R2."

specified value of `n`. The operations will continue as long as the stack is not empty, with a focus on examining the top element at each step. The stack will be progressively emptied through pop operations, ultimately returning the anticipated result. The `entryPoint` will enable us to manage the actions we wish to execute.

In this implementation, we have introduced the `CALL` case to address the base condition: when `n` equals 0, we return 1 and pop the stack. In cases where `n` is greater than 0, we compare the values of `i` and `n`, switch the `entryPoint` to `RESUME1`, and push a new context incorporating `i`. If neither condition is satisfied, we return the `return` value, which will contain the Catalan result and is applicable solely upon the completion of the Catalan number calculation.

Within the `RESUME1` case, the value of `return` is retained in `catalan1`, and the `entryPoint` is modified to `RESUME2`. A new context is created by decrementing `n` by `context.i - 1`. Subsequently, in `RESUME2`, we perform operations based on the value stored in `catalan2`, adding the product of `catalan1` and `return`, while incrementing the iterator `i`. A comparison is then conducted between `i` and `n`, similar to the approach taken in the `CALL` case. However, if the condition is satisfied, we simply update the `entryPoint` to `CALL`. Failing this, we manage the situation as in the `CALL` case by assigning the value of `c2` to `return` and executing a pop operation on the stack.

Ultimately, upon conclusion of the process, we return `return`, which will have accumulated the results of the operations conducted throughout the function.

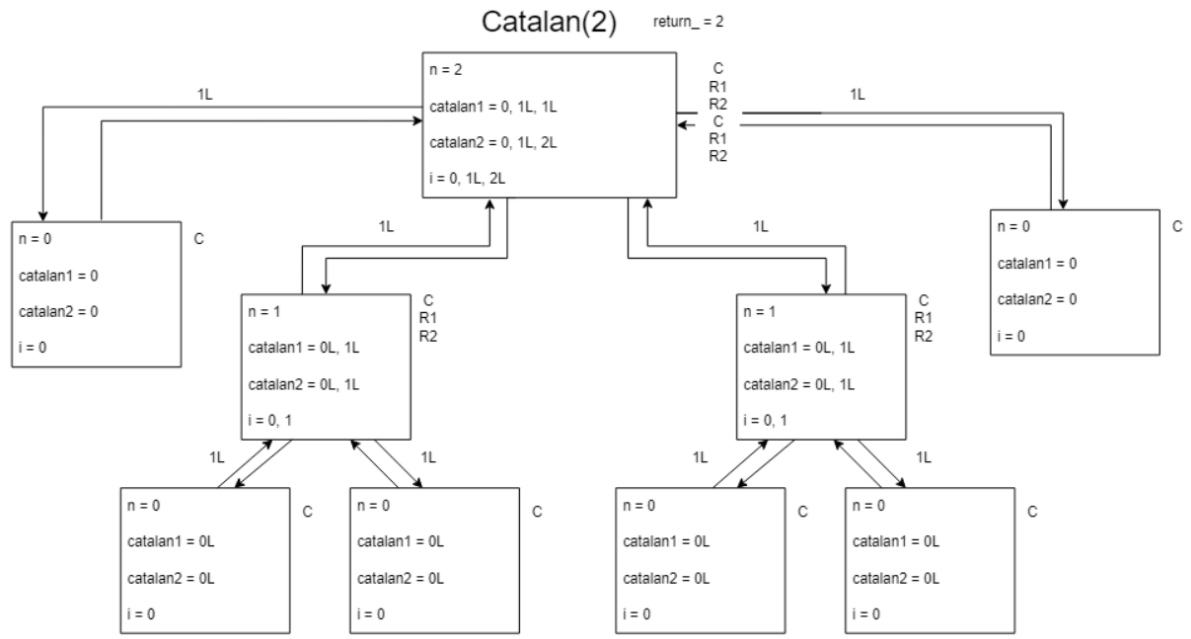

Figure 7: Tree of the process with Catalan (2) at the function `catalanIter`

```
01 | public static long catalanIter(int n) {
02 |     assert n >= 0 : "n must be greater or equal to 0";
03 |     long return_ = 0L;
04 |     var stack = new ArrayStack<Context>();
05 |     stack.push((new Context(n)));
06 |
07 |     while (!stack.isEmpty()) {
08 |         var context = stack.top();
09 |         switch (context.entryPoint) {
10 |             case CALL -> {
11 |                     if (context.n == 0) {
12 |                         return_ = 1L;
13 |                         stack.pop();
14 |                     } else if(context.i < context.n) {
15 |                         context.entryPoint = EntryPoint.RESUME1;
16 |                         stack.push(new Catalan.Context(context.i));
17 |                     } else {
18 |                         return_ = context.catalan2;
19 |                         stack.pop();
20 |                     }
21 |                 }
22 |
```

```
23 |             case RESUME1 -> {
24 |                     context.catalan1 = return_;
25 |                     context.entryPoint = EntryPoint.RESUME2;
26 |                     stack.push(new Catalan.Context(context.n - context.
   i - 1));
27 |                 }
28 |
29 |             case RESUME2 -> {
30 |                 context.catalan2 += context.catalan1 * return_;
31 |                 context.i++;
32 |                 if (context.i < context.n) context.entryPoint =
   EntryPoint.CALL;
33 |                 else {
34 |                         return_ = context.catalan2;
35 |                         stack.pop();
36 |                 }
37 |             }
38 |         }
39 |     }
40 |     return return_;
41 | }
```

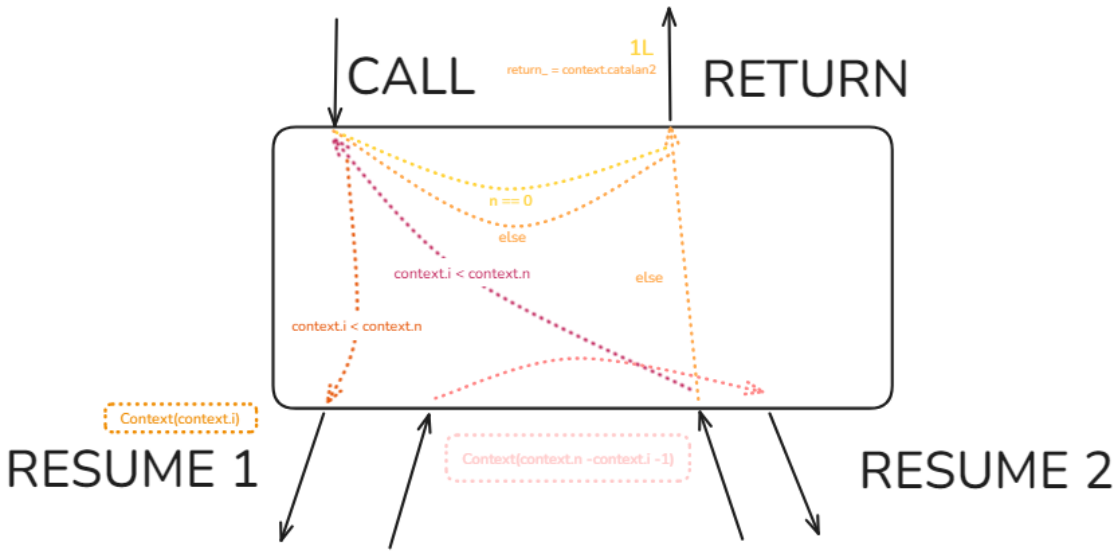

Figure 8: The diagram gives us a first look at how the cases are managed and how they communicate with each other

4 Conclusions

In summary, this practice has proven to be quite valuable, as it enabled us to deepen our understanding of stacks and explore the programming of their movements effectively. The tests introduced us to useful methods such as `snake_case`, which we can certainly apply in future projects. The hands-on experience of building and testing our own creations not only made the learning process enjoyable but also highlighted the importance of problem-solving techniques like using `assertEquals`.

Furthermore, our engagement with stack functionality was particularly rewarding, as it allowed us to apply it to mathematical concepts, including sequences like Catalan numbers. It was enlightening to observe how stacks can manage complex combinatorial ideas such as Fibonacci and factorial sequences. Although we initially found the schematic aspect challenging, it ultimately provided an intriguing opportunity to examine the internal dynamics of numbers, further enriching our overall learning experience.

Bibliography

[1] GeeksforGeeks. “Implement stack using array.” (Apr. 16, 2024), [Online]. Available: https://www.geeksforgeeks.org/implement-stack-using-array/. (accés: 15 de oct. de 2024).

[2] J. Gimeno. “Laboratorio 2 – pilas y la transformación recursivo a iterativo (v2).” (2024), [Online]. Available: https://cv.udl.cat/access/content/attachment/102010-2425/Activitats/d88d39bb-2b36-4eb6-ac29-11cc1612a273/Laboratorio%202%20-%20Enunciado%20_v2_.pdf. (accés: 20 de oct. de 2024).

[3] J. Gimeno. “Sequential data structures.” (Sep. 27, 2024), [Online]. Available: https://cv.udl.cat/access/content/group/102010-2425/Temario/3%20-%20Sequential%20Data%20Structures%20_v8_.pdf. (accés: 20 de oct. de 2024).