Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the MinStack class:

* MinStack() initializes the stack object.
* void push(int val) pushes the element val onto the stack.
* void pop() removes the element on the top of the stack.
* int top() gets the top element of the stack.
* int getMin() retrieves the minimum element in the stack.

You must implement a solution with O(1) time complexity for each function.

Example 1:

Input  
["MinStack","push","push","push","getMin","pop","top","getMin"]  
[[],[-2],[0],[-3],[],[],[],[]]

Output  
[null,null,null,null,-3,null,0,-2]Explanation  
MinStack minStack = new MinStack();  
minStack.push(-2);  
minStack.push(0);  
minStack.push(-3);  
minStack.getMin(); // return -3  
minStack.pop();  
minStack.top(); // return 0  
minStack.getMin(); // return -2

Constraints:

* -231 <= val <= 231 - 1
* Methods pop, top and getMin operations will always be called on non-empty stacks.
* At most 3 \* 104 calls will be made to push, pop, top, and getMin.

Here are two different functional ways to implement a MinStack in Scala:

1. Using two stacks:

This implementation uses two stacks as:

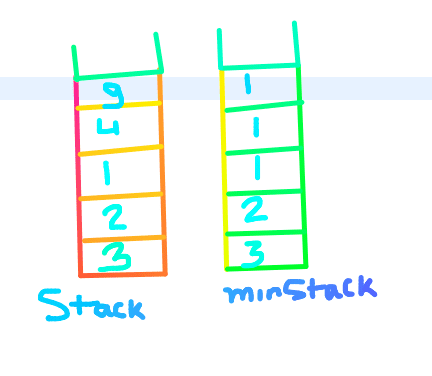
Stack 1: To store the actual elements.

Stack 2 : To store the minimum values encountered .

*Whenever a new element is pushed onto the stack, if it is smaller than or equal to the current minimum, it is also pushed onto the minStack.*

*When an element is popped from the stack, if it is the current minimum, it is also popped from the minStack.*

*The top of the minStack always stores the current minimum value.*



Using 2 stacks Implementation

class MinStack() {  
 private val stack = scala.collection.mutable.Stack[Int]()  
 private val minStack = scala.collection.mutable.Stack[Int]()  
  
 def push(x: Int): Unit = {  
 stack.push(x)  
 if (minStack.isEmpty || x <= minStack.top) minStack.push(x)  
 }  
  
 def pop(): Unit = {  
 if (stack.top == minStack.top) minStack.pop()  
 stack.pop()  
 }  
  
 def top(): Int = stack.top  
  
 def getMin(): Int = minStack.top  
}

2. Using a single stack with tuple

This implementation uses a single stack to store tuples of the form (element, minimum encountered).

*Whenever a new element is pushed onto the stack, its value and the current minimum are stored in a tuple and pushed onto the stack. The minimum value is updated whenever a new minimum is encountered.*

*When an element is popped from the stack, the corresponding tuple is also popped.*

*The top of the stack always stores the current element and the current minimum value.*

class MinStack() {  
 private val stack = scala.collection.mutable.Stack[(Int, Int)]()  
 private var min = Int.MaxValue  
  
 def push(x: Int): Unit = {  
 min = math.min(x, min)  
 stack.push((x, min))  
 }  
  
 def pop(): Unit = stack.pop()  
  
 def top(): Int = stack.top.\_1  
  
 def getMin(): Int = stack.top.\_2  
}

3. Using a single stack and a minimum variable

*We define a variable minEle that stores the current minimum element in the stack.*

*Now the interesting part is, how to handle the case when the minimum element is removed.?*

*We push “2x — minEle” into the stack instead of x so that the previous minimum element can be retrieved using the current minEle and its value stored in the stack.*

Follow the given steps to implement the stack operations:

Push(x): Insert x at the top of the stack

* If the stack is empty, insert x into the stack and make minEle equal to x.
* If the stack is not empty, compare x with minEle. Two cases arise:
* If x is greater than or equal to minEle, simply insert x.
* If x is less than minEle, insert (2\*x — minEle) into the stack and make minEle equal to x.  
  For example, let the previous minEle be 3. Now we want to insert 2. We update minEle as 2 and insert 2\*2–3 = 1 into the stack

Pop(): Removes an element from the top of the stack

* Remove the element from the top. Let the removed element be y. Two cases arise:
* If y is greater than or equal to minEle, the minimum element in the stack is still minEle.
* If y is less than minEle, the minimum element now becomes (2\*minEle — y), so update (minEle = 2\*minEle — y). This is where we retrieve the previous minimum from the current minimum and its value in the stack.  
  For example, let the element to be removed be 1 and minEle be 2. We remove 1 and update minEle as 2\*2–1 = 3

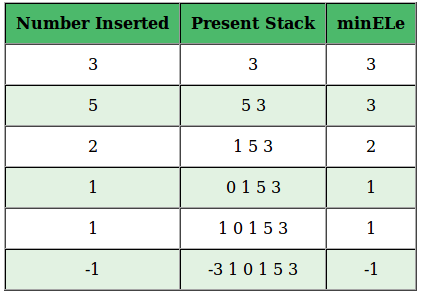
Important Points:

*Stack doesn’t hold the actual value of an element if it is minimum so far.*

*The actual minimum element is always stored in the minEle variable.*

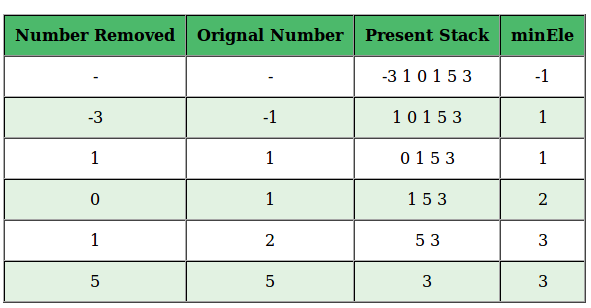
import java.util.Stack  
  
class MinStack {  
 private val s = new Stack[Node]  
  
 private class Node(val value: Int, val min: Int)  
  
 def push(x: Int): Unit = {  
 if (s.isEmpty()) {  
 s.push(new Node(x, x))  
 } else {  
 val min = Math.min(s.peek().min, x)  
 s.push(new Node(x, min))  
 }  
 }  
  
 def pop(): Int = {  
 s.pop().value  
 }  
  
 def top(): Int = {  
 s.peek().value  
 }  
  
 def getMin(): Int = {  
 s.peek().min  
 }  
}  
  
object MinStackDemo {  
 def main(args: Array[String]): Unit = {  
 val s = new MinStack()  
 // Function calls  
 s.push(-1)  
 s.push(10)  
 s.push(-4)  
 s.push(0)  
 println(s.getMin())  
 println(s.pop())  
 println(s.pop())  
 println(s.getMin())  
 }  
}

Push()



* *Number to be Inserted: 3, Stack is empty, so insert 3 into stack and minEle = 3.*
* *Number to be Inserted: 5, Stack is not empty, 5> minEle, insert 5 into stack and minEle = 3.*
* *Number to be Inserted: 2, Stack is not empty, 2< minEle, insert (2\*2–3 = 1) into stack and minEle = 2.*
* *Number to be Inserted: 1, Stack is not empty, 1< minEle, insert (2\*1–2 = 0) into stack and minEle = 1.*
* *Number to be Inserted: 1, Stack is not empty, 1 = minEle, insert 1 into stack and minEle = 1.*
* *Number to be Inserted: -1, Stack is not empty, -1 < minEle, insert (2\*-1–1 = -3) into stack and minEle = -1.*

Pop()



* *initially the minimum element minEle in the stack is -1.*
* *Number removed: -3, Since -3 is less than the minimum element the original number being removed is minEle which is -1, and the new minEle = 2\*-1 — (-3) = 1*
* *Number removed: 1, 1 == minEle, so number removed is 1 and minEle is still equal to 1.*
* *Number removed: 0, 0< minEle, original number is minEle which is 1 and new minEle = 2\*1–0 = 2.*
* *Number removed: 1, 1< minEle, original number is minEle which is 2 and new minEle = 2\*2–1 = 3.*
* *Number removed: 5, 5> minEle, original number is 5 and minEle is still 3*

Valid Parentheses In a string

*Leet code:* Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.
3. Every close bracket has a corresponding open bracket of the same type.

Example 1:

Input: s = "()"  
Output: true

Example 2:

Input: s = "()[]{}"  
Output: true

Example 3:

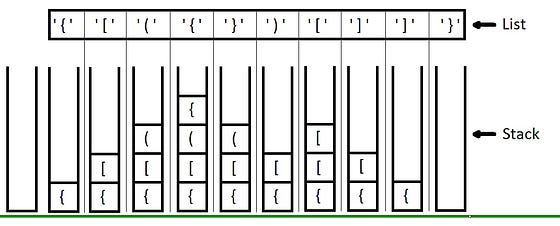
Input: s = "(]"  
Output: false

Constraints:

* 1 <= s.length <= 104
* s consists of parentheses only '()[]{}'.

If you look at the question , what we actually need is to validate the string with respect to parentheses(Open/Close).

Data structure : Stack.



Use stack to verify parentheses validation

Basic Impelmenattion approach : Lets create a stack of characters to keep track of the open brackets being encountered while traversing it. It then iterates through the string character by character, checking each character as follows:

* If the character is an opening bracket ((, [, or {), it is pushed onto the stack.
* If the character is a closing bracket (), ], or }), it is checked against the top of the stack. If the stack is empty or the top of the stack does not match the corresponding opening bracket, the string is not valid and the function returns false. Otherwise, the opening bracket is popped from the stack.
* If the character is neither an opening nor closing bracket, it is ignored.

After iterating through the entire string, if the stack is empty, the string is valid, otherwise it is not.

Lets try different ways to implement the same in scala:

1. Imperative Solution:

import scala.collection.mutable.Stack  
  
object ValidParentheses {  
 def isValid(s: String): Boolean = {  
 val stack = Stack[Char]()  
 val brackets = Map('(' -> ')', '{' -> '}', '[' -> ']')  
  
 for (c <- s) {  
 if (brackets.keySet.contains(c)) {  
 stack.push(c)  
 } else if (brackets.values.toSet.contains(c)) {  
 if (stack.isEmpty || brackets(stack.pop()) != c) {  
 return false  
 }  
 }  
 }  
  
 stack.isEmpty  
 }  
}

This program uses an imperative approach to process the string and a stack data structure to keep track of open brackets. The time complexity of this program is O(n), where n is the length of the input string, and the space complexity is O(n) for the stack data structure.

Pros: Efficient and easy to optimize for performance.

Cons: It is not as functional as the other approaches and may be harder to read and understand.

2. Recursive Solution:

object ValidParentheses {  
 def isValid(s: String): Boolean = {  
 def isMatched(c1: Char, c2: Char): Boolean =  
 (c1, c2) match {  
 case ('(', ')') | ('{', '}') | ('[', ']') => true  
 case \_ => false  
 }  
  
 def isValidHelper(s: List[Char], stack: List[Char]): Boolean =  
 (s, stack) match {  
 case (Nil, Nil) => true  
 case (h :: t, \_) if "({[".contains(h) => isValidHelper(t, h :: stack)  
 case (h :: t, sh :: st) if isMatched(sh, h) => isValidHelper(t, st)  
 case \_ => false  
 }  
  
 isValidHelper(s.toList, Nil)  
 }  
}

This program uses a recursive helper function to process the string and a stack data structure to keep track of open brackets. The time complexity of this program is O(n), where n is the length of the input string, and the space complexity is O(n) for the call stack and the stack data structure.

Pros: Simple and easy to understand recursive logic.

Cons: It may not be as efficient as other approaches due to recursion.

3. FoldLeft function:

object ValidParentheses {  
 def isValid(s: String): Boolean = {  
 val brackets = Map('(' -> ')', '{' -> '}', '[' -> ']')  
  
 s.foldLeft(List[Char]()) { (stack, c) =>  
 if (brackets.keySet.contains(c)) {  
 c :: stack  
 } else if (brackets.values.toSet.contains(c)) {  
 stack match {  
 case h :: t if brackets(h) == c => t  
 case \_ => return false  
 }  
 } else {  
 stack  
 }  
 }.isEmpty  
 }  
}

This program uses the foldLeft function to process the string and a list data structure to keep track of open brackets. The time complexity of this program is O(n), where n is the length of the input string, and the space complexity is O(n) for the list data structure.

Pros: Concise and easy to read functional code.

Cons: It may not be as efficient as imperative approaches due to the use of a list data structure.

Find if two strings are anagrams

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Problem: Given two strings s and t, return true *if* t *is an anagram of* s*, and* false *otherwise*.

**Example 1:**

Input: s = "anagram", t = "nagaram"  
Output: true

**Example 2:**

Input: s = "rat", t = "car"  
Output: false

**Constraints:**

* 1 <= s.length, t.length <= 5 \* 104
* s and t consist of lowercase English letters.

**Follow up:** What if the inputs contain Unicode characters? How would you adapt your solution to such a case?

Approaches:

1. There are several ways to check if two strings are anagrams in Scala. Here are three different approaches:

Sorting the strings One way to check if two strings are anagrams is to sort their characters and compare the results. If the two sorted strings are equal, then the original strings are anagrams. Here’s an example code:

def areAnagrams(s: String, t: String): Boolean = {  
 s.sorted == t.sorted  
}

This approach has a runtime complexity of O(n log n), where n is the length of the strings, since sorting requires O(n log n) time.

2. Using frequency counting

Another approach to check if two strings are anagrams is to use frequency counting.

We can create a frequency map of the characters in each string, and then compare the maps. If the two maps are equal, then the original strings are anagrams. Here’s an example code:

def areAnagrams(s: String, t: String): Boolean = {  
 if (s.length != t.length) return false  
   
 val sFrequency = s.groupBy(identity).mapValues(\_.length)  
 val tFrequency = t.groupBy(identity).mapValues(\_.length)  
   
 sFrequency == tFrequency  
}

This approach has a runtime complexity of O(n), where n is the length of the strings, since creating the frequency maps requires O(n) time.

3. Using XOR Another approach to check if two strings are anagrams is to use XOR operation. We can convert each character in the strings to its ASCII code and then XOR them together. If the XOR result is 0, then the original strings are anagrams. Here’s an example code:

def areAnagrams(s: String, t: String): Boolean = {  
 if (s.length != t.length) return false  
   
 val xorResult = s.zip(t).map { case (c1, c2) => c1.toInt ^ c2.toInt }.foldLeft(0)(\_ ^ \_)  
   
 xorResult == 0  
}

This approach also has a runtime complexity of O(n), where n is the length of the strings, since we’re iterating over the characters in the strings once to compute the XOR result.

However, this approach assumes that the strings only contain ASCII characters, so it may not work correctly for strings containing non-ASCII characters.

How will you solve a problem where you have both ASCII and NonASCII characters along with Unicode characters?

*we need to take into account that some Unicode characters may be represented using more than one Char in Scala.*

*To handle this case correctly, we need to normalize the strings using****Unicode normalization.***

Here’s an example code that uses Unicode normalization to find anagrams of two Unicode strings in Scala:

import java.text.Normalizer  
  
def areAnagrams(s: String, t: String): Boolean = {  
 val normalizedS = Normalizer.normalize(s, Normalizer.Form.NFC)  
 val normalizedT = Normalizer.normalize(t, Normalizer.Form.NFC)  
   
 if (normalizedS.length != normalizedT.length) return false  
   
 val sFrequency = normalizedS.groupBy(identity).mapValues(\_.length)  
 val tFrequency = normalizedT.groupBy(identity).mapValues(\_.length)  
   
 sFrequency == tFrequency  
}

In this code, we’re using the java.text.Normalizer class to normalize the two input strings using the Unicode normalization form NFC (composed normal form). This ensures that any composed Unicode characters are represented using a single Char in Scala.

After normalizing the strings, we’re using the same frequency counting approach as before to determine if they are anagrams.

Note that this approach still has a runtime complexity of O(n), where n is the length of the strings, since we’re still using groupBy and mapValues methods to count the frequency of characters in the strings.

Palindromic Number

Given an integer x, return true*if*x*is a****palindrome****, and*false*otherwise*.

**Example 1:**

Input: x = 121  
Output: true  
Explanation: 121 reads as 121 from left to right and from right to left.

**Example 2:**

Input: x = -121  
Output: false  
Explanation: From left to right, it reads -121. From right to left, it becomes 121-. Therefore it is not a palindrome.

**Example 3:**

Input: x = 10  
Output: false  
Explanation: Reads 01 from right to left. Therefore it is not a palindrome.

**Constraints:**

* -231 <= x <= 231 - 1

Here are some different functional ways to find if a number is palindromic in Scala with their respective time complexity:

1. Using string conversion and comparison (O(n)):

This approach converts the number to a string and compares it to its reverse. It has a time complexity of O(n), where n is the number of digits in the number.

def isPalindromic(num: Int): Boolean = {  
 num.toString == num.toString.reverse  
}

2. Using a recursive function (O(n)):

This approach uses a recursive function to build the reverse of the number and compare it to the original number. It has a time complexity of O(n), where n is the number of digits in the number.

def isPalindromic(num: Int): Boolean = {  
 def helper(n: Int, reversed: Int): Boolean = {  
 if (n == 0) reversed == num  
 else helper(n / 10, reversed \* 10 + n % 10)  
 }  
  
 helper(num, 0)  
}

3. Using a tail-recursive function (O(n)):

This approach uses a tail-recursive function, which is similar to the recursive function above but optimised for tail-call elimination. It also has a time complexity of O(n), where n is the number of digits in the number.

def isPalindromic(num: Int): Boolean = {  
 @annotation.tailrec  
 def helper(n: Int, reversed: Int): Boolean = {  
 if (n == 0) reversed == num  
 else helper(n / 10, reversed \* 10 + n % 10)  
 }  
 helper(num, 0)  
}

4. Using a while loop (O(n)):

This approach uses a while loop to build the reverse of the number and compare it to the original number. It has a time complexity of O(n), where n is the number of digits in the number.

def isPalindromic(num: Int): Boolean = {  
 var n = num  
 var reversed = 0  
 while (n > 0) {  
 reversed = reversed \* 10 + n % 10  
 n /= 10  
 }  
 num == reversed  
}