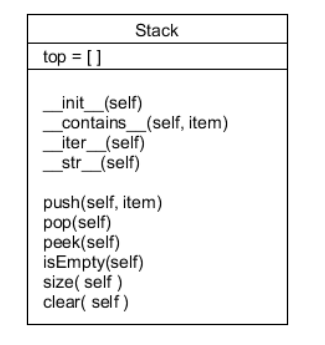
**Data Structures 2023-2**

**Lab 03: Stack Abstract Data Type**

**ID: 2021136150 Name: 양동재(Yang Dong Jae)**

1. **Task-1: Stack Abstract Data Type**

Write code for Stack abstract data type and test it

* 1. ****Object: Collection of data items such that last-in first out (LIFO) mechanism is maintained
  2. Operations
* push(x): adds an element x on the top of the stack
* pop(): removes the top element of the stack. The next element will become the top element
* isEmpty(): It returns true if the stack is empty, otherwise false
* peek(): It returns the top element without removing it from the Stack
* size(): It returns the number of items in the stack
* display(): It displays all the elements stored in the stack
* find(item): return true if item is found in the stack

**Code**

|  |
| --- |
| **Stack class src** |
| class Stack:  def \_\_init\_\_(self):  self.items = []  def \_\_str\_\_(self):  return str(self.items)  #return str(self.top)  def \_\_len\_\_(self):  return len(self.items)  def \_\_contains\_\_(self, item):  return item in self.items    def \_\_iter\_\_(self):  """Returns an iterator for the Stack."""  return iter(self.items)  def push(self, item):  """Adds an element 'item' on the top of the stack."""  self.items.append(item)  def pop(self):  """Removes the top element of the stack and returns it."""  if not self.isEmpty():  return self.items.pop()  else:  raise IndexError("Pop from an empty stack")  def peek(self):  """Returns the top element without removing it from the stack."""  if not self.isEmpty():  return self.items[-1]  else:  return None  def isEmpty(self):  """Returns True if the stack is empty, otherwise False."""  return len(self.items) == 0  def size(self):  """Returns the number of items in the stack."""  return len(self.items)  def display(self):  """Displays all the elements stored in the stack."""  print(self.items)  def find(self, item):  """Returns True if 'item' is found in the stack, otherwise False."""  return item in self.items  def clear(self):  """Clears all elements from the stack."""  self.items = []      class StackApplication(Stack):  def convert\_base(self, decimal, base):  """Converts a decimal number to binary, octal, or hexadecimal."""  stack = []  digits = "0123456789ABCDEF"  while decimal > 0:  remainder = decimal % base  stack.append(digits[remainder])  decimal //= base  result = ""  while stack:  result += stack.pop()  return result |

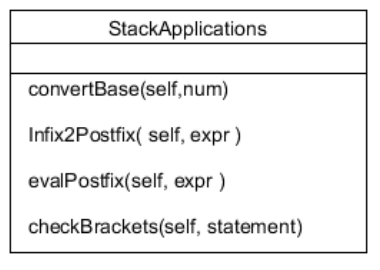
|  |
| --- |
| **Test code** |
| def useStack():  # Create an instance of the Stack class  stack = Stack()  # Test the push method  for i in range (20):  stack.push(i)  # Test the \_\_str\_\_ method  print("Stack:", stack)  # Test the size method  print("Size:", len(stack))  # Test the \_\_contains\_\_ method  print("Is 3 in stack?", 3 in stack)  # Test the pop method  popped\_item = stack.pop()  print("Popped item:", popped\_item)  # Test the peek method  top\_item = stack.peek()  print("Top item:", top\_item)  # Test the isEmpty method  print("Is stack empty?", stack.isEmpty())  # Test the display method  stack.display()  # Test the find method  print("Is 2 in stack?", stack.find(2))  # Test the clear method  stack.clear()  print("Stack after clearing:", stack) |

**Results/Output**

Insert pictures for the output of the programs written for this task

|  |
| --- |
| **Result of srouce code** |
| **Stack: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]**  **Size: 20**  **Is 3 in stack? True**  **Popped item: 19**  **Top item: 18**  **Is stack empty? False**  **[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]**  **Is 2 in stack? True**  **Stack after clearing: []** |

1. **Task-2: Applications of Stack data structure**

****Write code for the following applications of Stack data structure

* Convert Bases
* Evaluate postfix expressions
* Convert infix form to postfix form
* check brackets

**Code**

|  |
| --- |
| **Stack class src** |
| class StackApplication:  def \_\_init\_\_(self,stack):  self.stack = stack  def convertBase(self, num, base):  """Converts a decimal number to binary, octal, or hexadecimal."""  stack = self.stack  digits = "0123456789ABCDEF"    while num > 0:  remainder = num % base  stack.push(digits[remainder])  num //= base  result = ""  while not stack.isEmpty():  result += stack.pop()  return result  def Infix2Postfix(self, expr):  """Converts an infix expression to postfix."""  stack = self.stack  precedence = {'+': 1, '-': 1, '\*': 2, '/': 2, '^': 3}  postfix = []  for token in expr:  if token.isalnum():  postfix.append(token)  elif token == '(':  stack.push(token)  elif token == ')':  while not stack.isEmpty() and stack.peek() != '(':  postfix.append(stack.pop())  stack.pop() # Remove the '('  else:  while not stack.isEmpty() and stack.peek() != '(' and precedence.get(token, 0) <= precedence.get(stack.peek(), 0):  postfix.append(stack.pop())  stack.push(token)  while not stack.isEmpty():  postfix.append(stack.pop())  return ''.join(postfix)  def evalPostfix(self, expr):  """Evaluates a postfix expression."""  stack = self.stack  for token in expr:  if token.isdigit():  stack.push(int(token))  else:  operand2 = stack.pop()  operand1 = stack.pop()  if token == '+':  stack.push(operand1 + operand2)  elif token == '-':  stack.push(operand1 - operand2)  elif token == '\*':  stack.push(operand1 \* operand2)  elif token == '/':  stack.push(operand1 / operand2)  return stack.pop()  def checkBrackets(self, statement):  """Checks if the brackets in a statement are balanced."""  stack = self.stack  opening\_brackets = "([{"  closing\_brackets = ")]}"  for char in statement:  if char in opening\_brackets:  stack.push(char)  elif char in closing\_brackets:  if stack.isEmpty():  return False  top = stack.pop()  if opening\_brackets.index(top) != closing\_brackets.index(char):  return False  return stack.isEmpty() |

|  |
| --- |
| **Test code** |
| def useStackApplication():    odd = Stack()  for i in range (20):  odd.push(str(i))    app = StackApplication(odd)  # Testing convertBase method  decimal\_number = 42  binary = app.convertBase(decimal\_number, 2)  octal = app.convertBase(decimal\_number, 8)  hexadecimal = app.convertBase(decimal\_number, 16)  print(f"Decimal: {decimal\_number}")  print(f"Binary: {binary}")  print(f"Octal: {octal}")  print(f"Hexadecimal: {hexadecimal}")  # Testing Infix2Postfix method  infix\_expression = "3+4\*5"  postfix = app.Infix2Postfix(infix\_expression)  print(f"Infix: {infix\_expression}")  print(f"Postfix: {postfix}")  # Testing evalPostfix method  postfix\_expression = "34+5\*"  result = app.evalPostfix(postfix\_expression)  print(f"Result of {postfix\_expression} is {result}")  # Testing checkBrackets method  statement = "{[()()]}"  if app.checkBrackets(statement):  print(f"The brackets in {statement} are balanced.")  else:  print(f"The brackets in {statement} are not balanced.") |

**Results/Output**

Insert pictures for the output of the programs written for this task

|  |
| --- |
| **Result of srouce code** |
| **Decimal: 42**  **Binary: 101010191817161514131211109876543210**  **Octal: 52**  **Hexadecimal: 2A**  **Infix: 3+4\*5**  **Postfix: 345\*+**  **Result of 34+5\* is 35**  **The brackets in {[()()]} are balanced.** |

1. **Conclusion**

**1. Stack**

In this lectures, we learned about what is Stack in data structure, terminologies, consistent, etc.

At firstly, Stack Data Structure is child of list based linear data structure. And we can modified it for goals of that.

When an object is iterable, it means user can use it in a ‘for’ loop or with any iterator-related functions, like ‘iter()’ and ‘next()’

At first, Stack Data Structure is a child of list-based linear data structure. And also we can modify it depending on users' goals. So the Stack Data Structure is a defined link-based linear structure. Queue and list also belong of this category.

A diagram of a data structure

Description automatically generated

**1-1. Concepts**

Basically, a stack structure is type of list but it was gotten some restriction. So we can said it as restricted list. additionally we call restriction for list as “LIFO(Last In First Out)”. It means, when user add new item to stack, the new item will be placed at top, and also when remove data, it have to be removed an item that placed top of the stack. On this notion, we call it pushing when we add data to top of stack, and also popping when we remove data from top of stack. thus we can describe concepts of stack like this :

LIFO : Last In First Out

pushing : add data to top of stack

popping : remove data from top of stack

**1-2. Stack ADT (Abstract Data Type)**

Object : Collection of data items such that last-in first out mechanism is maintained

Operations

* Push
* Pop
* isEmpty : check stack is empty or not
* peek : return top element without remove data
* size
* display

but it is not an indispensable rule. As I told in from first phrase of the conclusion, a Stack is User user-modified data structure. That means you can implement the new method and remove, or modify it as far as you want.

**1-3. Application of Stack**

As I mentioned earlier, the main concept of stack ‘LIFO’, it is make a lot of positive effects. thus a lot of product, notion, algorithm are developing and will be developing based on stack data structure.

And below things are some of the important applications of stack :

* stacks can be used revers a word and string
* stacks can be used to check parenthesis matching in an expression
* stacks can be used in function calls.
* Stacks ca be used for expression evaluation.
* Stacks can be used for conversion form one form of expression to another
* Stacks can be used for memory management
* Stack data structures are used in backtracking problems.
* Stack can be used implement recursive problem

**1-4. Use of Stacks in Function Calls**

When we use the structure of a Stack, they have to be created to store the **current environment** And we call it **Activation Record(Stack Frame, Activation Stack)**.

Also, we can define the **current environment** as parameters, contents of registers, the function’s return value, local variable, and the address of the instruction to which execution is to return when the function finishes execution

**2. fragment of develop application**

In Python, the \_\_iter\_\_ method is a special method that we can define in our own classes to make instances of your class iterable. When an object is iterable, it means you can use it in a for loop or with any iterator-related functions, like iter() and next().

Here's how the \_\_iter\_\_ method works and how you can use it:

**Definition of** \_\_iter\_\_: To make your class iterable, you need to define the \_\_iter\_\_ method within the class. This method should return an iterator object, typically self (the instance of the class), and can also be used to perform any setup for iteration.

|  |  |
| --- | --- |
| When \_\_iter\_\_ function defined | When \_\_iter\_\_ function couldn’t defined |
| 0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | Type Error : 'Stack' object is not iterable |