**SSN College of Engineering**

Department of Information Technology

UIT2201 — Programming and Data Structures

2022 – 2023

**Exercise — 08**

U. Pranaav | IT-B | 3122225002093

I. AIM:

To provide an implementation of List ADT, using dynamically managed array with the following methods.

• Constructor, which takes an argument called ’val’ and creates a new list. If the argument ’val’ is

an integer, you should create an array of initial capacity ’val’. If the argument ’val’ is a sequence

(such as list, tuple, str, etc.) then your constructor should create an array of double the size of the

sequence ’val’ and initialize the newly created list with the elements of the sequence ’val’.

• A string method ( str() ) to convert this list object into a string.

• A length method ( len() ) that returns the size of this list object.

• A getitem() method to return the object stored at the given index.

• A setitem() method to update the object stored at the given index.

• An append method (append()), that adds the given object at the end if this list. Note that the

underlying array should be dynamically managed and there should not be any capacity limit.

• An insert method (insert()) to insert the given object at the given index. The size of this list

increases by 1, and there should not be any capacity limit. This list is mutated and no new list is

created.

• An extend utility (extend()) to extend the first list (’self’ if implemented as a method) with the

elements of the second list

• contains() functionality that returns ‘True’ is the given object is present in the list, ‘False’ otherwise. When implemented as a method, it should be named as contains ().

• index() functionality that returns the index of the first occurrence of the given object in the list.

An exception should be raised if the object is not present in the list.

• count() functionality that the returns the count of occurrences of the given object in the list.

To perform an empirical analysis to understand the time complexity of the ‘append()’ method. Write a function (takes an integer n as an argument) that creates an empty list and append n random objects to that list. Your function should record the time taken T for these n appends, and return the average time T/n. Run the experiment for different (very large) values of n and note down the average time taken per ‘append()’ operation.

II. CODE:

# -\*- coding: utf-8 -\*-

'''

This module provides a class used for creating an Array by

importing a module ctypes for creating a dynamic array.

This is a part of the excercises given under the course

UIT2201 (Programming and Data Structures).

In this source code I have executed my own logic. The code

follows good coding practices.

Your comments and suggestions are welcome.

Created on Wed May 24 2023

Revised on Wed May 25 2023

Original Author: U. Pranaav <pranaav2210205@ssn.edu.in>

'''

import ctypes

from timeit import default\_timer

*class* Array:

    '''

    The given class gives the implementation of a dynamic array

    in python.

    The input data is not modified in any way and there are

    no side effects.

    methods:

        \_\_init\_\_: the constructor

        \_\_setitem\_\_: for setting a certain value at an index

        \_\_getitem\_\_: for getting a value at an index

        \_\_len\_\_: for finding the length of array

        \_\_str\_\_: for displaying class objects in human readable

        form and return a string form of array.

        makearray: to create an array of a specified size.

        append: to add an element to the end of the array.

        resize: to increase size of array dynamically.

    '''

*def* \_\_init\_\_(*self*,*val*):

        '''

        A Array is a dynamic array implementation in Python that allows

        for efficient appending, indexing, and iteration of elements.

        Initializes a new instance of the Array class.

        Parameters:

        val: An integer value specifying the initial capacity of the Array.

        '''

        if isinstance(*val*, int):

*self*.n = 0

*self*.capacity = *val*

*self*.array = *self*.makearray(*val*)

        elif isinstance(*val*, list) or isinstance(*val*, tuple):

*self*.n = len(*val*)

*self*.capacity = len(*val*)

*self*.array = *val*

        elif isinstance(*val*, str):

*self*.n = len(*val*)

*self*.capacity = len(*val*)

*self*.array = [i for i in *val*]

*def* \_\_len\_\_(*self*):

        '''

        Returns the number of elements in the Array.

        '''

        return *self*.n

*def* makearray(*self*, *size*):

        '''

        Creates a new array of the specified size.

        Parameters:

        size: The size of the new array.

        Returns:

        A new array of the specified size.

        '''

        B = (*size* \* ctypes.py\_object)()

        return B

*def* append(*self*,*element*):

        '''

        Appends the specified element to the end of the Array.

        Parameters:

        element: The element to append to the end of the Array.

        '''

        if *self*.n == *self*.capacity:

*self*.array = *self*.resize(2 \* *self*.capacity)

*self*.array[*self*.n] = *element*

*self*.n += 1

*def* get\_capacity(*self*):

        """

        Returns the current capacity of the dynamic array.

        Returns:

            The current capacity of the array.

       """

        return *self*.capacity

*def* resize(*self*, *cap*):

        '''

        Resizes the Array to the specified capacity.

        Parameters:

        cap: The new capacity of the Array.

        Returns:

        The new capacity of the Array.

        '''

        new\_arr = *self*.makearray(*cap*)

*self*.capacity = *cap*

        for i in range(len(*self*.array)):

            new\_arr[i] = *self*.array[i]

        return new\_arr

*def* extend(*self*,*other*):

        '''

        Extends the Array by appending elements from another iterable.

        Parameters:

        other (iterable): An iterable containing elements to be appended.

        Returns:

        None

        Raises:

        None

        '''

        try:

            for i in *other*:

*self*.append(i)

        except:

            None

*def* insert(*self*,*ele*,*ind*):

        '''

        Inserts an element at the specified index in the Array.

        Parameters:

        ele: The element to be inserted.

        ind: The index at which the element should be inserted.

        Returns:

        None

        Raises:

        None

        '''

        if *self*.n > *ind*:

            for i in range(*self*.n,*ind*,-1):

*self*[i] = *self*[i-1]

*self*[*ind*] = *ele*

*def* delete(*self*, *idx*):

        """

        Deletes an element at a given index in the array.

        Args:

            idx: The index of the element to delete.

        Raises:

            IndexError: If the index is out of range

        """

        if not 0 <= *idx* < *self*.n:

            raise IndexError("Index out of range.")

        for i in range(*idx*, *self*.n - 1):

*self*.array[i] = *self*.array[i + 1]

*self*.n -= 1

        if *self*.n < *self*.capacity // 4:

*self*.resize(*self*.capacity // 2)

*def* index(*self*, *elt*):

        """

        Checks if an element is present in the array and returns the

        index of the element

        Args:

            elt: Element whose index is to be found out

        Returns:

            idx (int): If the element is found, index of the element

            is returned, else -1.

        """

        for idx in range(0, *self*.n):

            if *self*.array[idx] == *elt*:

                return idx

        return -1

*def* count(*self*, *count\_elt*):

        """

        Returns the number of occurrences of an element in the dynamic

        array

        Args:

            count\_elt: Element whose number of occurrences is to be found

        Returns:

            count (idx): The number of occurrences of an element in the

            dynamic array

        """

        count = 0

        for idx in range(0, *self*.n):

            if *self*.array[idx] == *count\_elt*:

                count += 1

        return count

*def* \_\_contains\_\_(*self*,*ele*):

        """

        Check if the given element is present in the array.

        Parameters:

        ele (Any): The element to check for presence in the array.

        Returns:

        bool: True if the element is found in the array, False

        otherwise.

        """

        try:

            for i in *self*.array:

                if *ele* == i:

                    return True

        except:

            return False

        return False

*def* \_\_getitem\_\_(*self*,*ind*):

        '''

        Gets the element at the specified index from the Array.

        Parameters:

        ind: The index of the element to get.

        Returns:

        The element at the specified index from the Array.

        '''

        return *self*.array[*ind*]

*def* \_\_setitem\_\_(*self*,*ind*,*ele*):

        '''

        Sets the element at the specified index in the Array to the

        specified value.

        Parameters:

        ind: The index of the element to set.

        ele: The new value for the element at the specified index.

        '''

        if *ind* <= *self*.n:

*self*.n += 1

*self*.array[*ind*] = *ele*

        else:

            raise IndexError("Index out of range")

*def* \_\_str\_\_(*self*):

        '''

        Converts the Array to a string.

        Returns:

        A string representation of the Array.

        '''

        out = '<'

        for i in range(*self*.n):

            try:

                if i != (*self*.n - 1):

                    out += str(*self*.array[i])

                    out += ','

                else:

                    out += str(*self*.array[i])

            except:

                continue

        return out + '>'

#driver code

if \_\_name\_\_ == '\_\_main\_\_':

    #this part of the code will only be run when the function is called directly

    #it will not be executed when it is imported as a module

    #initializing different sizes

    sizes = [1, 5, 10, 100, 500, 1000, 5000, 10000, 50000, 100000]

    #this loop calculates the times for appending until size gets extended

    for size in sizes:

        a = Array(size)

        start = default\_timer()

        for i in range(size+1):

            a.append(i)

        end = default\_timer()

        #amortized analysis

        print(*f*"SIZE : {size} \nTIME : {(end - start)} \nRATIO : {(end - start)/size} \n")

    lst1 = [x for x in range(1,11)]

    lst2 = [2 \* x for x in range(1,11)]

    print(*f*"List number 1 : {lst1}")

    print(*f*"List number 2 : {lst2}")

    print(*f*"Appending an element [5] to list 1: {lst1.append(5)}")

    new = Array(3)

    new[0] = 1

    new[1] = 900

    newone = Array(5)

    newone[0] = 500

    newone[1] = 400

    newone.append(1000)

    newone.extend(new)

    print(newone)

    print(1000 in newone)

    print(9000 in newone)

    print(newone)

    print(newone.count(1000))

    print(newone.index(500))

    newone.delete(0)

    print(newone)

    print(newone.get\_capacity())

III. OUTPUT:

SIZE : 1

TIME : 6.429999999998937e-05

RATIO : 6.429999999998937e-05

SIZE : 5

TIME : 9.389999999999399e-05

RATIO : 1.8779999999998797e-05

SIZE : 10

TIME : 0.0001507000000000036

RATIO : 1.507000000000036e-05

SIZE : 100

TIME : 0.0003811999999999982

RATIO : 3.811999999999982e-06

SIZE : 500

TIME : 0.0020791000000000004

RATIO : 4.158200000000001e-06

SIZE : 1000

TIME : 0.0025705999999999923

RATIO : 2.5705999999999924e-06

SIZE : 5000

TIME : 0.015021700000000013

RATIO : 3.0043400000000026e-06

SIZE : 10000

TIME : 0.0377748

RATIO : 3.7774799999999997e-06

SIZE : 50000

TIME : 0.14513179999999998

RATIO : 2.9026359999999997e-06

SIZE : 100000

TIME : 0.2825943

RATIO : 2.825943e-06

List number 1 : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

List number 2 : [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

Appending an element [5] to list 1: None

<500,400,1000,1,900>

True

False

<500,400,1000,1,900>

1

0

<400,1000,1,900>

5