NYU, Tandon School of Engineering

CS-UY 1134 Data Structures and Algorithms — Summer 2024

Homework #3
Due by Friday 6/14, 11:59pm

Submission instructions:

1. For each coding question, submit a .py file, name it using the pattern 'YourNetID_hwX_qX.py'.
2. For those questions that requires a written answer, submit a .pdf file for each question, name it using the pattern 'YourNetID_hwX_qX.pdf'.
   If a question has more than one sub-question that requires a written answer, put them in one .pdf file.

Note: your netID follows an abc123 pattern, not N12345678.

You should submit your homework via Gradescope.
For Gradescope's autograding feature to work:
a. Name all classes, functions and methods **exactly as they are in the assignment specifications**.
b. Make sure there are **no print statements** in your code. If you have tester code, please put it in a "main" function and **do not call it**.

Question 1:

Consider the implementation we made in class for `ArrayList`, and its extensions you did in the lab.

In this question, we will add two more methods to this class: the `insert` method and the `pop` method. For this question, please submit the modified `ArrayList` class.

a. Implement the method `insert(self, index, val)` that inserts `val` before `index` (shifting the elements to make room for `val`).

   For example, your implementation should follow the behavior below:

```
>>> arr_lst = ArrayList()
>>> for i in range(1, 4+1):
... arr_lst.append(i)
>>> arr_lst.insert(2, 30)
>>> arr_lst
[1, 2, 30, 3, 4]
```

   Notes:

   1. Make sure to double the capacity of the array, if there is no room for an additional element.
   2. Your function should raise an `IndexError` exception in any case the index (positive or negative) is out of range.

b. Implement the method `pop(self)`, that removes the last element from the list. The `pop` method should return the element removed from the list, and put `None` in its place in the array. If `pop` was called on an empty list, an `IndexError` exception should be raised.

   In order to maintain the linear memory usage of the list data structure, and its efficient amortized performance, we use the following shrinking strategy: When the number of elements in the list goes strictly below a quarter of the array's capacity, we shrink its capacity by half.

   For example, your implementation should follow the behavior below:

```
>>> arr_lst = ArrayList()
>>> for i in range(1, 5+1):
... arr_lst.append(i)
```

```
>>> arr_lst.pop()
5
>>> arr_lst.pop()
4
>>> arr_lst.pop()
3
>>> arr_lst.pop()
2
>>> arr_lst
[1]
```

Note: After the executing the code above, the capacity of the array in `ArrayList` will be 4.

c. *Extra Credit (You don't need to submit)*: The extending and shrinking strategies we use in our `ArrayList` implementation (doubling the capacity of the array each time there is no room to add an element, and shrinking the capacity of the array by a factor of 2 each time the number of elements is less than a quarter of the array's capacity), guarantees two important things:

   i. In any given time, the memory used to store the elements is linear. That is, if there are n elements in the array, then:
   $n \leq (capacity\ of\ the\ array) \leq 4n$.

   ii. Any sequence of n `append` and/or `pop` operations on an initially empty `ArrayList` object, takes $O(n)$ time.

   Proving these properties is out of the scope of this assignment, but we will show two supporting insights:

   1. Show that the following series of $2n$ operations takes $O(n)$ time: n `append` operations on an initially empty array, followed by n `pop` operations.
   2. Consider a variant to our shrinking strategy, in which an array of capacity N, is resized to capacity precisely that of the number of elements, any time the number of elements in the array goes strictly below $N/2$.
   Show that there exists a sequence of n `append` and/or `pop` operations on an initially empty `ArrayList` object, that requires $\Omega(n^2)$ time to execute.

d. Modify the `pop` method, so it could optionally get an index as a parameter. This index indicates the position of the element that is to be removed from the list.

Notes:
1. Make sure to use the same shrinking strategy described above in section (b).
2. Your function should raise an `IndexError` exception in any case the index (positive or negative) is out of range.

Question 2:

The `remove(value)` method of the `list` class, removes the **first** occurrence of `value` from the list it was called on, or raises a `ValueError` exception, if `value` is not present.
Note: Since `remove` needs to shift elements, its worst-case running time is **linear.**

In this question we will look into the function `remove_all(lst, value)`, that removes **all** occurrences of `value` from `lst`.

a) Consider the following implementation of `remove_all`:
```
def remove_all(lst, value):
 end = False
 while(end == False):
 try:
 lst.remove(value)
 except ValueError:
 end = True
```

Analyze the worst-case running time of the implementation above.

b) Give an implementation to `remove_all` that runs in **worst-case linear time**.
Notes:
1. Your implementation should **mutate the given list object** (in-place), without using an additional data structure.
2. Your implementation **should keep the relative order** of the elements that remain in the list

c) Analyze the worst-case running time of your implementation in (b).

Question 3:

Analyze the running time of each of the following functions. For each function: i. Draw the recursion tree that represents the execution process, and the cost of each call.
ii. Conclude the total (asymptotic) running time.

Note: For the simplicity of the analysis of sections (b) and (c), you may assume that `n` is a power of 2, therefore it can always be divided evenly by 2.

a.

```
def fun1(n):
  if (n == 0):
      return 1
  else:
      part1 = fun1(n-1)
      part2 = fun1(n-1)
      res = part1 + part2
      return res
```

b.

```
def fun2(n):
  if (n == 0):
      return 1
  else:
      res = fun2(n//2)
      res += n
      return res
```

c.

```
def fun3(n):
  if (n == 0):
      return 1
  else:
      res = fun3(n//2)
      for i in range(1, n+1):
          res += i
      return res
```

Question 4:

Give a **recursive** implement to the following functions:

a) **def** `count_lowercase(s, low, high):`

The function is given a string `s`, and two indices: `low` and `high` (`low` ≤ `high`), which indicate the range of indices that need to be considered.

The function should return the number of lowercase letters at the positions *low, low+1, …, high* in `s`.

b) def `is_number_of_lowercase_even(s, low, high):` The function is given a string `s`, and two indices: `low` and `high` (`low` ≤ `high`), which indicate the range of indices that need to be considered.

The function should return `True` if there are even number of lowercase letters at the positions *low, low+1, …, high* in `s`, or `False` otherwise.

Question 5:

Give a **recursive** implement to the following function:

def `split_by_sign(lst, low, high)`

The function is given a list `lst` of non-zero integers, and two indices: `low` and `high` (`low` ≤ `high`), which indicate the range of indices that need to be considered. The function should reorder the elements in `lst`, so that all the negative numbers would come before all the positive numbers.

Note: The order in which the negative elements are at the end, and the order in which the positive are at the end, doesn't matter, as long as all he negative are before all the positive.

Question 6:

A ***nested list of integers*** is a list that stores integers in some hierarchy. That is, its elements are integers and/or other nested lists of integers.
For example `nested_lst=[[1, 2], 3, [4, [5, 6, [7], 8]]]` is a nested list of integers.

Give a **recursive** implement to the following function:

```
def flat_list(nested_lst, low, high)
```

The function is given a nested list of integers `nested_list`, and two indices: `low` and `high` (`low` ≤ `high`), which indicate the range of indices that need to be considered.

The function should flatten the sub-list at the positions *low, low+1, …, high* of `nested_list`, and return this flattened list. That is, the function should create a new 1-level (non hierarchical) list that contains all the integers from the *low…high* range in the input list.

For example, when calling `flat_list` to flatten the list `nested_lst` demonstrated above (the initial call passes `low=0` and `high=2`), it should create and return `[1, 2, 3, 4, 5, 6, 7, 8]`.