Lecture 1: Linear Arrays

[CSE220 | Data Structures | STV/SFT]

1. Linear Array Properties

1.1. All the elements will be of the same data type

1.2. All the non-dummy values will be at the left followed by the dummy values (zero/None/null)

1.3 Length of an array is fixed once declared and cannot be changed

2. Ways of Creating a Numpy Array

```python
import numpy as np
arr = np.array([0]*5)
print(arr)
print(type(arr))

arr1 = np.array([1, 2, 3, 4])
print(arr1)
print(type(arr1))

arr2 = np.zeros(6, dtype=int)
print(arr2)
print(type(arr2))
```

```
[0 0 0 0 0]
<class 'numpy.ndarray'>
[1 2 3 4]
<class 'numpy.ndarray'>
[0 0 0 0 0 0]
<class 'numpy.ndarray'>
```

3. Array Operations

3.1 Iteration

3.2 Resize

3.3 Copy (Pass by Value)

3.4 Shift: Left and Right

3.5 Rotate: Left and Right

3.6 Reverse: Out-of-place and In-place

3.7 Insert: At the end or anywhere else

3.8 Delete: Last element or any other element

3.1 Iteration

Iteration refers to checking all the values index by index. The main idea is to go to that memory location and check all the values index by index.

```python
arr1 = np.array([1, 2, 3, 4])

print("Using Regular For Loop")
for i in arr1:
  print(i)

print("Using Ranged For Loop")
for i in range(0, len(arr1),1):
  print(arr1[i])
```

```
Using Regular For Loop
1
2
3
4
Using Ranged For Loop
1
2
3
4
```

3.2 Resize

We can not resize an array because it has a fixed memory location. However, if we ever need to resize an array, we need to create a new array with a new length and then copy the values from the original array. For example, if we have an array [10, 20, 30, 40, 50] whose length is 5 and want to resize the array with length 8. The new array will be [10, 20, 30, 40, 50, None, None, None].

```python
import numpy as np

def resizeArray(arr, new_size):
  arr2= np.array([0]* new_size)
  for i in range(len(arr)):
    arr2[i]= arr[i]
  return arr2

arr1= np.array([1,2,3,4])
print(resizeArray(arr1, 6))
# x= resizeArray(arr1, 6)
```

```
[1 2 3 4 0 0]
```

3.3 Copy (Pass by Value)

Copy array means you initialize a new array with the same length as the given array to copy and then copy the old array's value by value. As the variable where we store the array only stores the memory location, only copying the value is not enough for array copying. For example, if you have an array titled arr = [1, 2, 3, 4] and write a2 = arr. It does not mean you have copied arr to a2.

```
arr1= np.array([1,2,3,4])
arr2= arr1  #pass by reference
print(id(arr1)) #id() function is used to find the memory location
print(id(arr2))
```

```
139056372101744
139056372101744
```

```
def copyArray(arr):
  arr2= np.array([0]* len(arr))
  for i in range(len(arr)):
    arr2[i]= arr[i]
  return arr2

arr1= np.array([1,2,3,4])
arr2= copyArray(arr1)
# arr2= arr1.copy()
print(id(arr1))
print(id(arr2))
```

```
136491905968784
136491905966672
```

3.4 Shift: Left and Right

Shifting an Array Left: Shifting an entire array left moves each element one (or more, depending how the shift amount) position to the left. Obviously, the first element in the array will fall off at the beginning and be lost forever. The last slot of the array before the shift (ie. the slot where the last element was until the shift) is now unused (we can put a None there to signify that). For example, shifting the array [5, 3, 9, 13, 2] left by one position will result in the array [3, 9, 13, 2, None]. Note how the array[0] element with the value of 5 is now lost, and there is an empty slot at the end.

```
def shiftLeft(arr):
  for i in range(1, len(arr), 1):
    arr[i-1]= arr[i]
  arr[len(arr)-1]= 0
  return arr

arr= np.array([1,2,3,4])
print(shiftLeft(arr))
```

```
[2 3 4 0]
```

Shifting an Array Right: Shifting an entire array right moves each element one (or more, depending how the shift amount) position to the right. Obviously, the last element in the array will fall off at the end and be lost forever. The first slot of the array before the shift (ie., the slot where the first element was until the shift) is now unused (we can put a None there to signify that). The size of the array remains the same however because the assumption is that you would something in the now-unused slot. For example, shifting the array [5, 3, 9, 13, 2] right by one position will result in the array [None, 5, 3, 9, 13]. Note how the array[4] element with the value of 2 is now lost, and there is an empty slot at the beginning.

```
def shiftRight(arr):
  for i in range(len(arr)-1, 0, -1):
    arr[i]= arr[i-1]
  arr[0]=0
  return arr

arr= np.array([1,2,3,4])
print(shiftRight(arr))
```

```
[0 1 2 3]
```

3.5 Rotate: Left and Right

Rotating an Array Left: Rotating an array left is equivalent to shifting a circular or cyclic array left where the 1st element will not be lost, but rather move to the last slot. Rotating the array [5, 3, 9, 13, 2] left by one position will result in the array [3, 9, 13, 2, 5].

```
def rotateLeft(arr):
  temp= arr[0]
  for i in range(1, len(arr), 1):
    arr[i-1]= arr[i]
  arr[len(arr)-1]= temp
  return arr

arr= np.array([1,2,3,4])
print(rotateLeft(arr))
```

```
[2 3 4 1]
```

Rotating an Array Right: Rotating an array right is equivalent to shifting a circular or cyclic array right where the last element will not be lost, but rather move to the 1st slot. Rotating the array [5, 3, 9, 13, 2] right by one position will result in the array [2, 5, 3, 9, 13].

```
def rotateRight(arr):
  temp= arr[len(arr)-1]
  for i in range(len(arr)-1, 0, -1):
    arr[i]= arr[i-1]
  arr[0]= temp
  return arr

arr= np.array([1,2,3,4])
print(rotateRight(arr))
```

```
[4 1 2 3]
```

3.6 Reverse: Out-of-place and In-place

Reversing an array can be implemented in two ways. First, we will create a new array with the same size as the original array and then copy the values in reverse order. The method is called out-of-the-place operation.

However, an efficient approach might be to reverse the array in the original array. By this, we will not need to allocate extra spaces. This is known as an in-place operation. To do so we need to start swapping values from the beginning position to the end position. The idea is to swap starting value with the end value, then the second value with the second last value, and so on.

```
def reverse_out_of_place(arr):
  arr2= np.zeros(len(arr), dtype=int)
  i= 0
  while(i<=len(arr)-1):
    arr2[i]= arr[len(arr)-1-i]
    i+=1

  return arr2

arr= np.array([1,2,3,4])
print(reverse_out_of_place(arr))
arr= np.array([1,2,3,4,5])
print(reverse_out_of_place(arr))
```

```
[4 3 2 1]
[5 4 3 2 1]
```

```
def reverse_in_place(arr):
  x, j = 0, len(arr)-1
  for i in range(len(arr)//2):
    arr[i], arr[j]= arr[j], arr[i]
    x+=0
    j-=1
  return arr

arr= np.array([1,2,3,4])
print(reverse_in_place(arr))
arr= np.array([1,2,3,4,5])
print(reverse_in_place(arr))
```

```
[4 3 2 1]
[5 4 3 2 1]
```

Introducting New Linear Array Property: Size

- Size indicates the number of non-dummy values in an array.
- Size of an array can never exceed the length of that array. Which means size > length is not possible.
- Size cannot be negative.
- Size is used in array insertion or deletion, or any task that involves working with the non-dummy values only.
- If size < length, insertion in an array is possible, otherwise must resize the array.
- If size > 0, at least one element from the array can be deleted, otherwise, there is nothing to delete.

```
#Complete the function print_non_dummies that prints all the non-dummy values of an array
import numpy as np
```

```
def print_non_dummies(arr, size):
  for i in range(0, size):
    print(arr[i])

arr= np.zeros(6, dtype=int)
arr[0], arr[1], arr[2] = 4, 0, 5
print(arr)
print_non_dummies(arr, 3) #array, 3
```

```
[4 0 5 0 0 0]
4
0
5
```

3.7 Insert: At the end or anywhere else

- At first, check whether insertion is possible or not. If not possible, resize the array
- Valid places of inserting: index 0 to size
- If you are inserting at the end of the array, no shifting is needed
- If you are inserting anywhere else but at the end of an array, then the subsequent elements must be right shifted before insertion
- Increase size after insertion

```
#Inserting at the end
def insert_at_the_end(arr, size, elem):
  if size >= len(arr):
    arr= resizeArray(arr, len(arr)+5)

  arr[size]= elem
  return arr

arr= np.zeros(4, dtype=int)
arr[0], arr[1], arr[2] = 4, 6, 5
print(arr)
print(insert_at_the_end(arr, 3, 15)) #array, size, elem
print(insert_at_the_end(arr, 4, 25)) #array, size, elem
```

```
[4 6 5 0]
[ 4  6  5 15]
[ 4  6  5 15 25  0  0]
```

```
#Inserting anywhere
def insert_anywhere(arr, size, index, elem):
  if index<0 or index>size:
    return "Insertion Not Possible"

  if size >= len(arr):
    arr= resizeArray(arr, len(arr)+3)

  for i in range(size, index, -1): #Right Shift
    arr[i]= arr[i-1]

  arr[index]= elem
  return arr

arr= np.zeros(4, dtype=int)
arr[0], arr[1], arr[2] = 4, 6, 5
print(arr)
print(insert_anywhere(arr, 3, 1, 15)) #array, size, index, elem
print(insert_anywhere(arr, 4, 4, 25)) #array, size, index, elem, the size increased

# If you want to insert at the end with this function too,
# all you have to do is provide the index value equal to the size
```

```
[4 6 5 0]
[ 4 15  6  5]
[ 4 15  6  5 25  0  0]
```

3.8 Remove: Last element or any other element

- At first, check whether deletion is possible or not
- Valid places of deletion: index 0 to size-1
- If you are deleting the last element of the array, no shifting is needed
- If you are deleting any other element except for the last element, then the subsequent elements must be left shifted after deletion
- Decrease size after deletion

```
#Deleting last element
def delete_last_element (arr, size):
  if size == 0:
    return "Deletion Not Possible"

  arr[size-1]= 0
```

```python
  return arr

arr= np.zeros(4, dtype=int)
arr[0], arr[1], arr[2] = 4, 6, 5
print(arr)
print(delete_last_element(arr, 3)) #array, size
print(delete_last_element(arr, 2)) #array, size
```

```
[4 6 5 0]
[4 6 0 0]
[4 0 0 0]
```

```python
#Deleting any element
def delete_any_element (arr, size, index):
  if size == 0 or (index<0 and index>=size):
    return "Deletion Not Possible"

  for i in range(index, size, 1):
    arr[i]=arr[i+1]
  arr[size-1]=0

  return arr

arr= np.zeros(5, dtype=int)
arr[0], arr[1], arr[2], arr[3] = 4, 6, 5, 2
print(arr)
print(delete_any_element(arr, 4, 0)) #array, size, index
print(delete_any_element(arr, 3, 2)) #array, size, index
```

```
[4 6 5 2 0]
[6 5 2 0 0]
[6 5 0 0 0]
```

Homework 1: Check Palindrome

Look up what a palindrome is. Write a function that checks whether an array is a plaindrome or not.

```python
def palindrome(arr, size):
  pass

arr= np.zeros(5, dtype=int)
arr[0], arr[1], arr[2], arr[3] = 4, 6, 6, 4
print(palindrome(arr, 4)) #array, size

arr= np.zeros(5, dtype=int)
arr[0], arr[1], arr[2] = 4, 6, 4
print(palindrome(arr, 3)) #array, size

arr= np.zeros(5, dtype=int)
arr[0], arr[1], arr[2], arr[3] = 4, 6, 4, 5
print(palindrome(arr, 4)) #array, size
```

```
True
True
False
```

Homework 2: Reverse Print

Write a function Rev_Print that Reverse iterates and prints all the non-dummy values of an array.

Function Call:

arr= np.array([1 3 2 5 0 0])

print(Rev_Print(arr, 4)) #Array, size

Output:

5 2 3 1

Homework 3: Merge Two Arrays

Write a function Merge_Arrays that takes two arrays and the sizes of those two arrays (total 4 parameters) and mergers the two arrays. There will be no dummy value in the merged array.

Function Call:

arr1= np.array([1 5 0 0])

arr2= np.array([3 5 2 0])

print(Merge_Arrays(arr1, 2, arr2, 3)) #Array1, Array1 size, Array2, Array2 size

Output:

[1 5 3 5 2]