List Vs Array

There are two basic ways of storing the sequence of values:

List	**Array**
Flexible Length	Fixed Size
Easy to modify the structure	Support `random access`
Values scattered in memory	Allocates `contiguous` block of memory
`Sequence of nodes`	
Each node contains value and points to next node in sequence `linked list`	`index-value` storage thus random access
Access: takes time proportional to index the elem is located `O(n)`	Access: `constant` time independent of index
Insert/Delete: `Constant`	Insert/Delete: `O(n)`
Swap: `O(n)`	Swap: `Constant`
Good for `Insertion Sort`	Good for `Binary Search`
	List in Python are arrays

Implementing Lists in pyton:

```
class Node:
  def __init__(self, v=None):
    self.value = v    #self.value: None-> empty list
    self.next = None  #points to next node
    return

  def isempty(self):
      if self.value == None:
        return(True)
      else:
        return(False)
```

```
# create lists
l1 = Node() # empty list
l2 = Node(5) # Singletone list
```

appending to the list

```
def append(self, v):
  if self.isempty():
    self.value = v
  elif self.next == None:
    self.next = Node(v)
  else:
    self.next.append(v)
  return
```

```
def appendi(self, v):
  # append, iterate
  if self.isempty():
    self.value = v
    return

    temp = self
    while temp.next != None;
      temp = temp.next

    temp.next = Node(v)
    return
```

insert at start of list

```
def insert(self, v):
  if self.isempty():
    self.value = v
    return

  newnode = Node(v)

  # Exchange values in self and newnode
  (self.value,newnode.value) = (newnode.value, self.value)
  # switch the links:
  (self.next, newnode.next) = (newnode, self.next)

  return
```

Remove first occurrence of v

- Scan for first v- look ahead at next node
- If next node is v: Bypass it!

```
def delete(self, v):
  if self.isempty():
    return

  if self.value == v:
    self.value = None
    if self.next != None:
      self.value = self.next.value
      self.next = self.next.next
    return
  else:
    if self.next != None:
      self.next.delete(v)
      if self.next.value == None:
        self.next = None
```

summary

- Use linked list of nodes to implement a flexible list
- Append is easy
- Insert requires some care, cannot change where the head points to
- When deleting, look on step ahead to bypass the node to be deleted.

List in Python

- `Arrays` - `double space` as and when needed
- Keep track of last position of list in the array
    - `l.append()` and `l.pop()` are `constant time`, amortised `O(1)` CHEAP
    - Insertion and deletion requires `O(n)` EXPENSIVE
- Useful for representing the matrices
- Need to be careful when initialising multidimensional list:
    - `zero_list = [0,0,0]`
    - `zero_matrix = [zero_list, zero_list, zero_list]`
    - `zero_matrix[1][1] = 1`
    - `[[0,1,0],[0,1,0],[0,1,0]]` Mutability
    - Solution:
        - list comprehension
            - `[[O for i in range(3)]for j in range(3)]`
        - numpy library

```
import numpy as np
zeromatrix = np.zeros(shape =(3,3))
```

Dictionary in Python

Array/List	Dictionary
Access: Positional Indices	Access: `Arbitrary Keys`
	`Random Access`
	Implemented as hash tables

- Underlying storage is `array`
- Keys have to be mapped to {0,1,…,n-1}
  - `Hash Function` : `h: S → X` maps a set of values S to a small range of integers X = {0,1,2,—-,n-1}
  - Typically `|X| << |S|` so there will be collisions, `h(s) = h(s′) , s ≠ s′`
  - A good hash function will minimise collisions
  - `SHA-256` is an industry standard for the Hashing function whose range is 256 bits
  - lets say `d = dict()`
    - `d[k] = v`
    - hash: `k –> h(k)`
    - put v at h(k) position
- How to avoid the collisions:

Open Addressing[Closed Hashing]	**open Hashing**
Probe a `sequence` of alternate slots in **`same array`**	Each slot in the array points to a list of values
Example: Parking lot: Look for the open slot of parking.	Each position in array store the values in list
	`List` for each position