Data Structures

Presented by Yasin Ceran

January 18, 2024

Data Structure Interfaces

- A data structure is a way to store data, with algorithms that support operations on the data.
- Collection of supported operations is called an interface (also API or ADT).
- Interface is a specification: what operations are supported (the problem!).
- Data structure is a representation: how operations are supported (the solution!).
- In this class, two main interfaces: Sequence and Set.

Sequence Interface

- Maintain a sequence of items (order is extrinsic).
- Example: $(x_0, x_1, x_2, \ldots, x_{n-1})$ (zero indexing).
- (use n to denote the number of items stored in the data structure).
- Supports sequence operations:

Container	`build(X)` `len(n)`	given an iterable `X`, build sequence from items in `X` return the number of stored items

Static	`iter_seq()`
	`get_at(i)`
	`set_at(i,x)`
		return the stored items one-by-one in sequence order
		return the i^{th} item
		replace the i^{th} item with x

Dynamic	`insert_at(i,x)`
	`delete_at(i)`
	`insert_first(x)`
	`delete_first(x)`
	`insert_last(x)`
	`delete_last(x)`
		add x as the i^{th} item
		remove and return the the i^{th} item
		add x as the first item
		remove and return the first item
		add x as the last item
		remove and return the last item

Table: Sequence Operations

- special case interfaces:

stack	`insert_last(x)` and `delete_last(x)`
queue	`insert_last(x)` and `delete_first(x)`

Set Interface

- Sequence about extrinsic order, set is about intrinsic order
- Maintain a set of items having unique keys (e.g., item x has key $x.key$)
- Often we let key of an item be the item itself, but may want to store more info than just key
- Supports set operations:

Container	`build(X)` `len(n)`	given an iterable `X`, build sequence from items in `X` return the number of stored items
Static	`find(k)`	return the stored items iwth key k
Dynamic	`insert(x)` `delete(k)`	add x to set (replace item with $x.key$ if one already exists) remove and return the stored item with key k

Order	`iter_ord()`
	`find_min()`
	`find_max()`
	`find_next(k)`
	`find_prev(k)`
		return the stored items one-by-one in key order
		return the stored item with smallest key
		return the stored item with largest key
		return the stored item with smallest key larger than k
		return the stored item with largest key smaller than k

Table: Set Operations

- special case interfaces:

dictionary	set without the Order operations

Array Sequence

- Array is great for static operations! `get_at(i)` and `set_at(i, x)` in $\mathcal{O}(1)$ time!
- But not so great at dynamic operations...
- (For consistency, we maintain the invariant that array is full)
- Then inserting and removing items requires:
  - reallocating the array
  - shifting all items after the modified item

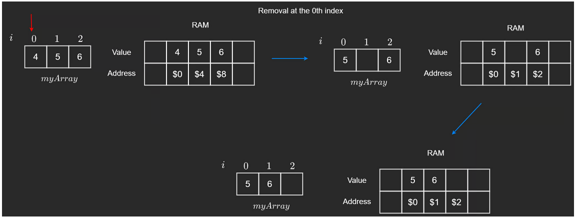
Data Structure	Container	Static	Dynamic		
	`build(X)`	`get_at(i)` `set_at(i)`	`insert_first(x)` `delete_first(x)`	`insert_last(x)` `delete_last(x)`	`insert_at(i,x)` `delete_at(i,x)`
Array	n	1	n	n	n

Table: Array Operations-Worst Case $\mathcal{O}$

Array Sequence-Remove First

- Given the target index i, we can iterate from $i+1$ until the end of the array and shift each element 1 position to the left. In the worst case, we will need to shift all of the elements to the left.

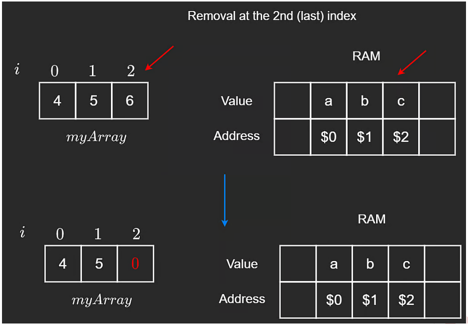
```python
# Remove value at index i before shifting elements to the left.
# Assuming i is a valid index.
def removeMiddle(arr, i, length):
    # Shift starting from i + 1 to end.
    for index in range(i + 1, length):
        arr[index - 1] = arr[index]
    # No need to 'remove' arr[i], since we already shifted
```



Array Sequence-Remove Last

- When we want to remove an element from the last index of an array, setting its value to 0 / null or -1 does the job. While it is not being "deleted" per se, this overwriting denotes an empty index. We will also reduce the length by 1 since we have one less element in the array after deletion.

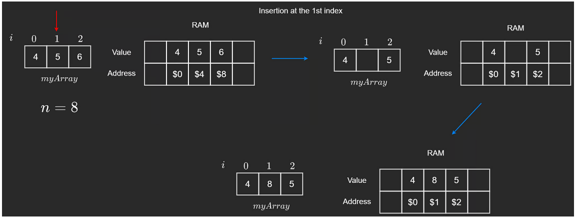
```python
# Remove from the last position in the array if the array
# is not empty (i.e. length is non-zero).
def removeEnd(arr, length):
    if length > 0:
        # Overwrite last element with some default value.
        # We would also consider the length to be decreased by 1.
        arr[length - 1] = 0
```



Array Sequence-Insert at the ith index

- To insert, we will need to shift all values one position to the right.

```python
# Insert n into index i after shifting elements to the right.
# Assuming i is a valid index and arr is not full.
def insertMiddle(arr, i, n, length):
    # Shift starting from the end to i.
    for index in range(length - 1, i - 1, -1):
        arr[index + 1] = arr[index]

    # Insert at i
    arr[i] = n
```



Array Sequence-Code-I

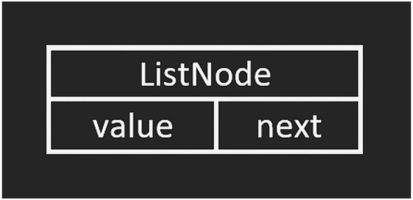
```
class Array_Seq:
    def __init__(self):                     #O(1)
        self.A = []
        self.size = 0

    def __len__(self):
        return self.size                    #O(1)

    def __iter__(self):
        yield from self.A                   #O(n) iter_seq

    def build(self, X):
        self.A = [a for a in X]   # pretend this builds a static array
        self.size = len(self.A)

    def get_at(self, i):
        return self.A[i]                     #O(1)

    def set_at(self, i, x):
        self.A[i] = x                        #O(1)

    def _copy_forward(self, i, n, A, j):     #O(n)
        for k in range(n):
            A[j + k] = self.A[i + k]

    def _copy_backward(self, i, n, A, j):     #O(n)
        for k in range(n - 1, -1, -1):
            A[j + k] = self.A[i + k]
```

Array Sequence-Code-2

```python

    def insert_at(self, i, x):                      #O(n)
        n = len(self)
        A = [None] * (n + 1)
        self._copy_forward(0, i, A, 0)
        A[i] = x
        self._copy_forward(i, n - i, A, i + 1)
        self.build(A)

    def delete_at(self, i):                         #O(n)
        n = len(self)
        A = [None] * (n - 1)
        self._copy_forward(0, i, A, 0)
        x = self.A[i]
        self._copy_forward(i + 1, n - i - 1, A, i)
        self.build(A)
        return x

    def insert_first(self, x):                      #O(n)
        self.insert_at(0, x)

    def delete_first(self):                         #O(n)
        return self.delete_at(0)

    def insert_last(self, x):                       #O(n)
        self.insert_at(len(self), x)

    def delete_last(self):                          #O(n)
        return self.delete_at(len(self) - 1)
```

Linked List Sequence

- Pointer data structure (this is not related to a Python "list").
- Each item stored in a node which contains a pointer to the next node in sequence.
- Each node has two fields: `node.item` and `node.next`.
  - `node.item` stores the value of the node, the value can be anything - a character, an integer, etc.
  - `node.next` stores the reference to the next node in the linked list.

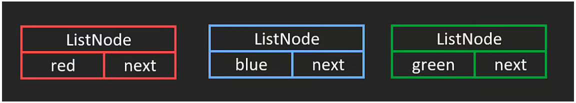


Creating a Linked List from scratch

- Chaining `ListNode` objects together is what results in a linked list. Creating your own `ListNode` class would look like the following in code.

```
class ListNode:
    def __init__(self, val):
        self.val = val
        self.next = None
```

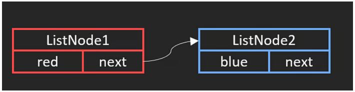
- Suppose that we have three `ListNode` objects – `ListNode1`, `ListNode2`, `ListNode3`, and we instantiate them with the following values as seen in the visual below.



Linking the Nodes

- Can manipulate nodes simply by relinking pointers!
- Upon instantiation, the nodes would be stored in an arbitrary order in the memory.
- Maintain pointers to the first node in sequence (called the head).
- Next, we would need to make sure that our next pointers point to another `ListNode`, and not `null`.

```
ListNode1.next = ListNode2
ListNode2.next = null
```



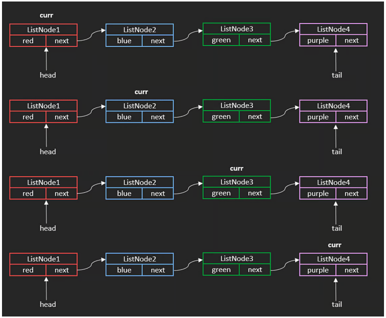
- Since `ListNode2` is the last node in the `LinkedList`, its next pointer will point to `null`.
- Can now insert and delete from the front in $\mathcal{O}(1)$ time!
- Inserting/deleting efficiently from back is also possible.
- But now `get_at(i)` and `set_at(i, x)` each take $\mathcal{O}(n)$ time.

Traversal

- To traverse a linked list from beginning to end, we can just make use of a simple while loop.

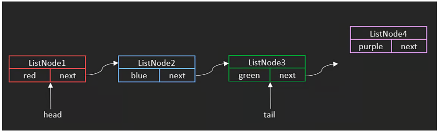
```
cur = ListNode1
while cur:
    cur = cur.next
```

- We start the traversal at the beginning, `ListNode1`, and assign it to a variable `cur`, denoting the current node we are at.
- We keep running the while loop and updating our `cur` to the next node until we encounter a node that is `null` — meaning we are at the end of the linked list and traversal is finished.
- Traversal is $\mathcal{O}(n)$



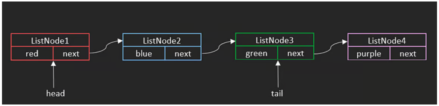
Inserting to the Tail in Linked List

- We want to add a new node, `ListNode4` to the end of a linked list



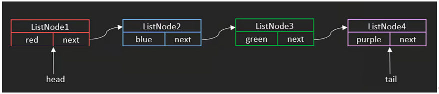
- Once `ListNode4` is appended, we update our tail pointer to be at `ListNode4`.

```
tail.next = ListNode4
```



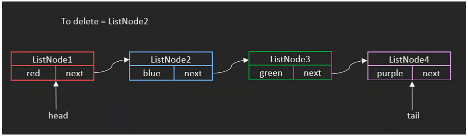
- We then update the tail pointer `ListNode4`.

```
tail = ListNode4
```

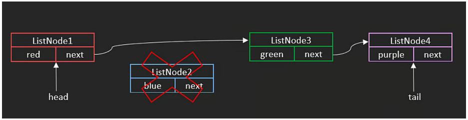


Deleting from a Linked List

- We want to delete `ListNode2`




```
head.next = head.next.next
```



- Updated linked list after deletion of `ListNode2`. Notice that now `ListNode1`'s next pointer points to `ListNode3`, instead of `ListNode2`.

Linked List Sequence-Summary

- Can manipulate nodes simply by relinking pointers!
- Maintain pointers to the first node in sequence (called the head).
- Can now insert and delete from the front in $\mathcal{O}(1)$ time!
- Inserting/deleting efficiently from back is also possible.
- But now `get at(i)` and `set at(i, x)` each take $\mathcal{O}(n)$ time.

Data Structure	Container	Static	Dynamic		
	`build(X)`	`get_at(i)` `set_at(i,x)`	`insert_first(x)` `delete_first(x)`	`insert_last(x)` `delete_last(x)`	`insert_at(i,x` `delete_at(i,x`
Linked List	n	n	1	n	n

Table: Linked List Operations-Worst Case $\mathcal{O}$

Linked List-Code-1

```

class Linked_List_Seq:          # O(1)
    def __init__(self):
        self.head = None
        self.size = 0

    def __len__(self):          # O(1)
        return self.size

    def __iter__(self):         # O(n) iter_seq
        node = self.head
        while node:
            yield node.item
            node = node.next

    def build(self, X):         # O(n)
        for a in reversed(X):
            self.insert_first(a)

    def get_at(self, i):          # O(i)
        node = self.head.later_node(i)
        return node.item
```

Linked List-Code-2

```python
    def set_at(self, i, x):          #O(i)
        node = self.head.later_node(i)
        node.item = x

    def insert_first(self, x):          #O(1)
        new_node = Linked_List_Node(x)
        new_node.next = self.head
        self.head = new_node
        self.size += 1

    def delete_first(self):          #O(1)
        x = self.head.item
        self.head = self.head.next
        self.size -= 1
        return x

    def insert_at(self, i, x):          #O(i)
        if i == 0:
            self.insert_first(x)
            return
        new_node = Linked_List_Node(x)
        node = self.head.later_node(i - 1)
        new_node.next = node.next
        node.next = new_node
        self.size += 1
```

Linked List-Code-3

```
    def delete_at(self, i):          #O(i)
        if i == 0:
            return self.delete_first()
        node = self.head.later_node(i - 1)
        x = node.next.item
        node.next = node.next.next
        self.size -= 1
        return x

    def insert_last(self, x):          #O(n)
        self.insert_at(len(self), x)

    def delete_last(self):          #O(n)
        return self.delete_at(len(self) - 1)
```

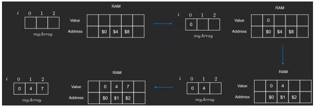
Dynamic Array Sequence

- Make an array efficient for last dynamic operations.
- Python "list" is a dynamic array.
- Idea: Allocate extra space so reallocation does not occur with every dynamic operation.
- Fill ratio: $0 \leq r \leq 1$ the ratio of items to space.
- Whenever array is full ($r = 1$), allocate $\mathcal{O}(n)$ extra space at end to fill ratio r_i (e.g., 1/2).
- Will have to insert $\mathcal{O}(n)$ items before the next reallocation.
- A single operation can take $\mathcal{O}(n)$ time for reallocation.
- However, any sequence of $\mathcal{O}(n)$ operations takes $\mathcal{O}(n)$ time.
- So each operation takes $\mathcal{O}(1)$ time "on average".

Mechanics of dynamic arrays

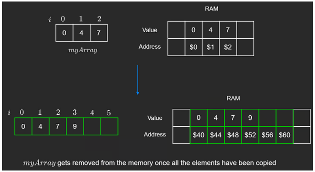
- When inserting into a dynamic array, the operating system finds the next empty space and pushes the element into it

```
# Insert n in the last position of the array
def pushback(self, n):
    if self.length == self.capacity:
        self.resize()

    # insert at next empty position
    self.arr[self.length] = n
    self.length += 1
```



- Since the array is dynamic, adding another element when we run out of capacity is achieved by copying over the values to a new array that is double the original size

```
def resize(self):
   # Create new array of double capacity
   self.capacity = 2 * self.capacity
   newArr = [0] * self.capacity

   # Copy elements to newArr
   for i in range(self.length):
       newArr[i] = self.arr[i]
   self.arr = newArr
```



Amortized Analysis

- Data structure analysis technique to distribute cost over many operations.
- Operation has amortized cost $T(n)$ if k operations cost at most $\leq kT(n)$.
- "$T(n)$ amortized" roughly means $T(n)$ "on average" over many operations.
- Inserting into a dynamic array takes $\mathcal{O}(1)$ amortized time.
- Amortized time complexity is the average time taken per operation, that once it happens, it won't happen again for so long that the cost becomes "amortized". This makes sense because it is not always that the array needs to be resized.

Array-Linked List Sequence-Dynamic Array

Data Structure	Container	Static	Dynamic		
	`build(X)`	`get_at(i)` `set_at(i)`	`insert_first(x)` `delete_first(x)`	`insert_last(x)` `delete_last(x)`	`insert_at(i,x)` `delete_at(i,x)`
Array	n	1	n	n	n
Linked List	n	n	1	n	n
Dynamic Array	n	1	n	$1_{(a)}$	n

Table: Array, Linked List, Dynamic Array Operations-Worst Case $\mathcal{O}$

Dynamic Array-Code-1

```
class Dynamic_Array_Seq(Array_Seq):
    def __init__(self, r = 2):        #O(1)
        super().__init__()
        self.size = 0
        self.r = r
        self._compute_bounds()
        self._resize(0)

    def __len__(self):        #O(1)
        return self.size

    def __iter__(self):        #O(n)
        for i in range(len(self)):
            yield self.A[i]

    def build(self, X):        #O(n)
        for a in X:
            self.insert_last(a)

    def _compute_bounds(self):        #O(1)
        self.upper = len(self.A)
        self.lower = len(self.A) // (self.r * self.r)
```

Dynamic Array-Code-2

```
def _resize(self, n):        #O(1) or O(n)
    if (self.lower < n < self.upper):
        return
    m = max(n, 1) * self.r
    A = [None]*m
    self._copy_forward(0, self.size, A, 0)
    self.A = A
    self._compute_bounds()

def insert_last(self, x):        #O(1)a
    self._resize(self.size + 1)
    self.A[self.size] = x
    self.size += 1


def delete_last(self):         #O(1)a
    self.A[self.size - 1] = None
    self.size -= 1
    self._resize(self.size)
```

Dynamic Array-Code-3

```
    def insert_at(self, i, x):       #O(n)
        self.insert_last(None)
        self._copy_backward(i, self.size - (i + 1), self.A, i + 1)
        self.A[i] = x

    def delete_at(self, i):       #O(n)
        x = self.A[i]
        self._copy_forward(i + 1, self.size - (i + 1), self.A, i)
        self.delete_last()
        return x

    def insert_first(self, x):       #O(n)
        self.insert_at(0, x)

    def delete_first(self):       #O(n)
        return self.delete_at(0)
```