Data Structures (Abstraction)

- ADT: Abstract Data Type

Arrays

- A type of ADT for storing and accessing data in computer's memory
- Array Data Structure:
    - Homogenous data: Represents a sequence of elements of the same data type
    - Elements can be accessed, retrieved, stored or replaced at given index positions
- Random Access and Contiguous Memory:
    - Array indexing is a random access operation
    - Address of item: base address + offset
        - Index operation:
            1. Fetch base address of array's memory block
            2. Find target index by adding offset (Index * k) to base address, k is number of memory cells required by an array item
- Advantage:
    - Access element directly (e.g. A[11] accesses 12th element)
    - Supported by many languages
    - Use less memory than linked structure
- Disadvantage:
    - A lot of data movement during operation e.g. adding/ removing elements
    - Static, must declare array size, inflexible when more data is needed

- **One-Dimensional Array**
    - General form:

      *mylist = [ ] * MAXSIZE*

    - Index from 1 to MAXSIZE ensuring total number of elements = MAXSIZE
    - OR Index from 0 to MAXSIZE-1
        - Element name: [], e.g. scores[4]

- **for Loop** to read values into array:
  - General form:

```
mylist = [ ]
for i in range(SIZE):
    value = input("enter value")
    mylist.append( value )
```

- **while Loop** to process data more than once:
  - Read list up to X number of elements
  - Using input -999 to end

```
mylist = [ ]
value = input("enter value")
while value is not "-999" or value is not -999:
    mylist.append( value )
```

- **Parallel Arrays:**
  - Two or more arrays with same size and index

```
mylist1 = [ None for i in range(SIZE) ]
mylist2 = [ None for i in range(SIZE) ]

for i in range(SIZE):
    value1 = input("enter value 1: ")
    value2 = input("enter value 2: ")
    mylist1[ i ], mylist2[ i ] = value1, value2
```

- **Two-Dimensional Array (Matrix / Grid)**
  - Used for information that fits naturally into a table
  - Two subscripts are necessary to specify an element in a matrix
    - Row subscript
    - Column subscript
  - General form:

```
matrix = [ [ None ] for i in range(COLUMN) ] for j in range(ROW)
                                   # initialise 2D array
matrix[0][0] = value               # assign value
```

```
for row in matrix:                        # output in 2D format
    for column in row:
        print(column, end = " ")
    print( )
```

Dictionary

- A type of randomly accessed data structure
- Contains a key and a corresponding value associated with it
- Keys are unique in each dictionary, there isn't any duplicate keys
- Values aren't unique, multiple keys can correspond to the same value
- Initialise dictionary:

```
dictionary = { }
dictionary["abc"] = 101          # assign value to key ( dictionary[k] = v )
```

- Dictionary methods (built-in):
    - Get value of specified key (returns value):
      dictionary.get("abc")
      dictionary["abc"]

    - Get all dictionary keys (returns array):
      dictionary.keys( )

    - Get all dictionary values (returns array):
      dictionary.values( )

    - Get all dictionary items (returns array):
      dictionary.items( )

    - Remove specified key (and value) from dictionary:
      dictionary.pop("abc")   # error if specified element isn't in dictionary

    - Check if key / value is in dictionary (returns boolean):
      if k in dict.keys( ): ...
      if v in dict.values( ): ...

- Sorting elements in a dictionary:

- Dictionaries aren't automatically sorted
- Returning values from dictionary may not be in the original/ chronological order
- Built-in sorting functions:
  - By key:

```
sorted_elements = [ ]
for key in sorted( dictionary.keys( ) ):
      sorted_elements.append( key, dictionary[key] )
```

  - By value:

```
sorted_elements = [ ]
for k, v in dictionary.items( ):
      sorted_elements.append([k, v])
sorted_elements.sort( key = lambda x: x[1] )
```

- Manually sorting:

```
      sorted_elements = [ ]
      while len( dictionary.keys( ) ) is not 0:
# loops if dictionary isn't empty
            smallest_key, smallest_val = None, None

            for k, v in dictionary.items( ):
# traverse dictionary to find smallest key / val
                  if ( smallest_key == None ) or ( k < smallest_key):
# or v < smallest_val if sorting by value
                        smallest_key, smallest_val = k, v
            sorted_elements.append( [smallest_key, smallest_val] )
            dictionary.pop( smallest_key )
# remove smallest key from dictionary
```

Linked Structure

- A type of structure that requires traversal, not randomly accessed
- Types:
  - Singly linked: Linking in one direction
  - Doubly linked: Linking in both directions
- Basic unit of representation: Node

    - Singly linked node contains:
        - Data value
        - Pointer to the next node
- Methods to set up singly linked array:
    - Two parallel arrays
    - Pointers
        - Specified value (e.g. None or 0) represents end of structure
        - Python: "None" can be used as empty link
- Start pointer points to the first node
- End node with "null" as next value to indicate the end of linked list
- Defining singly linked node class:
    - Flexibility and ease of use are critical
    - Node instance variables are usually referenced without method calls
    - Constructors allow user to set a node's link when node is created

```
class Node(object):
    def __init__ (self, data, next = None):
        self.data = data
        self.next = next
```

- Using singly linked node class:
    - Node variables are initialised to "None" or a new "Node" object

```
node1 = None      # empty link
node2 = Node ( "A", None )      #node containing data and empty link (End node)
node3 = Node ( "B", node2 )    # node containing data and a link to node2
```

- Operations on singly linked structures:
    - Traversal:
        - Visit each node without deleting it
        - Uses a temporary pointer variable

```
curr = head
while curr != None:          # None serves as sentinel to stop
```

process

```
<use or modify curr.data>
curr = curr.next
```

- Searching:
  - Resembles a traversal
  - Two possible sentinels;
    1. Empty link
    2. Data item that equals to target item

```
curr = head
while ( curr != None ) and ( curr.data != targetItem ):
    curr = curr.next
if curr != None:
    <target has been found>
else:
    <target is not in linked structure>
```

- Accessing *i*th item is sequential structure:

```
# Assume  0 <= i < n
# i is item to be accessed
# n is number of nodes in structure

curr = head
while i > 0:
    curr = curr.next
    i = i - 1
return curr.data
```

- Traversal to access specified value:
  - If target item not present: no replacement occurs, operation returns False
  - If target item is present: new item replaces it, operation returns True

```
curr = head
while ( curr != None ) and ( target != curr.data ):
    curr = curr.next
if curr != None:
    return curr.data
else:                    # not found
```

```
        return None
```

- Replacing *i*th item:

```
# Assume  0 <= i < n:
# i is item to be processed
# n is number of nodes in structure

curr = head
while i >  0:
     curr = curr.next
     i = i - 1
curr.data = newItem
```

- Inserting at beginning:

```
newNode = Node(newItem)
newNode.next = head.next
head.next = newNode
```

- Advantage:
  - Little data movement, no shifting of items needed for insertion or removal, only need to change pointer
- Disadvantage:
  - No random access, must traverse list

Binary Search Tree   # notes incoming

-

Stacks

- FILO (first-in-last-out) / LIFO (last-in-first-out) structure
  - First item to be pushed into the stack is the last to be removed
  - Last item to be pushed into the stack is the first to be removed
- Access is completely restricted to just one end, called the "top"
- Stack methods:
  - self.push(item): inserts item at top of stack
  - self.pop( ): removes and returns item at the top of stack, if stack is not empty

    - self.peek( ): returns item at top of stack, if stack is not empty, does not remove item
    - self.isEmpty( ): returns True if stack is empty, otherwise False
    - self.__len__( ): returns the number of elements in the stack currently, same as len(s)
    - self.__str__( ): returns the string representation of the stack, same as str(s)
- Applications of Stacks:
    - Evaluating arithmetic expressions:
        - Infix form: operator located between operands (A+B)
            - Sometimes require parentheses
            - Involves rules of mathematical operation
            - E.g. (33 + 22 * 2 = 33 + 44 = 78)
        - Postfix form: operator immediately follows operands (A B +)
            - Does not require parentheses
            - Evaluation applies operators as soon as they are encountered
            - E.g. (34 22 2 * + = 34 44 + = 78)
    - Evaluating Infix to Postfix:
        1. Start with empty postfix expressions and empty stack (stack holds operators and left parentheses)
        2. Scan across infix expressions from left to right
        3. Encounter operand: append to postfix expression
        4. Encounter '(': push into stack
        5. Encounter operator:
            1. Pop all operators from stack with equal or higher precedence
            2. Append to postfix expression
            3. Push scanned operator onto stack
        6. Encounter ')': pop all operators from stack to postfix expression until meeting matching '(', discard '('
        7. Encounter end of infix expression: pop remaining operators from stack to postfix expression
    - Evaluating Postfix:
        1. Scan across postfix expression from left to right
        2. Encounter operator: apply it to two preceding operands, replace all three expressions with result
        3. Continue scanning until the end of expression has been reached

```
Create new stack
While postfix expression is not empty:
    Get next token
    If token is operand:
        Push token into stack
    Else:
        If token is operator:
            Pop the top two operands from stack
            Apply operator to the two operands
            Push resulting value into stack
    EndIf
EndWhile
Return final value
```

- Memory management:
  - Keeps track of details during programme runtime
    - When a function calls another function, it interrupts its own execution and needs to be able to resume its excurion in the same state it was in when it was interrupted
    - LIFO behaviour
  - When a function is called:
    1. Push a copy of its activation record onto the runtime stack
    2. Copy its arguments into parameter spaces
    3. Transfer control to starting address of the body of function
    - Top actication record in the runtime stack is always the function that's currently being executed
  - When a function terminates:
    1. Pop activastion record or terminated function from runtime stack
    2. Use new top activation record to restore environment of interrupted function and resume execution of the interrupted function
- Implementation:
  - Using array:

```
class ArrayStack:
    CAPACITY = 100
```

```
def __init__(self):
    self.items = list( )
    for i in range( ArrayStack.CAPACITY ):
        self.items.append( None )
    self.top = -1
    self.size = 0

def push(self, newItem):
    if self.isFull( ):
        print("Stack is full")
        return ""
    else:
        self.size += 1
        self.top += 1
        self.items[self.top] = newItem

def pop(self):
    if self.isEmpty( ):
        print("Stack is empty")
        return ""
    else:
        oldItem = self.items[self.top]
        self.top -= 1
        self.size -= 1
        return oldItem

def peek(self):
    if self.isEmpty( ):
        print("Stack is empty")
        return ""
    else:
        return self.items[self.top]

def __len__(self):
    return self.size

def isEmpty(self):
    return self.size == 0

def isFull(self):
    return self.size == ArrayStack.CAPACITY
```

```
    def __str__(self):
        res = ""
        for i in range( len(self) ):
            res = res + str( self.items[i] ) + " "
        return res
```

- Using linked structure:

```
class Node:
    def __init__(self, data, next):
        self.data = data
        self.next = next

class LinkedStack:
    def __init__(self):
        self.top = None
        self.size = 0

    def push(self, newItem):
        newNode = Node( newItem, self.top )
        self.top = newNode
        self.size += 1

    def pop(self):
        if self.isEmpty( ):
            print("Stack is empty")
            return ""
        else:
            oldItem = self.top.data
            self.top = self.top.next
            self.size -= 1
            return oldItem

    def peek(self):
        if self.isEmpty( ):
            print("Stack is empty")
            return ""
        else:
            return self.top.data
```

```
def __len__(self):
    return self.size

def isEmpty(self):
    return self.size == 0

def __str__(self):
    res = ""
    curr = self.top
    while curr != None:
        res += str( self.top.data ) + " "
        curr = curr.next
        return res
```

Queue

- Insertion is restricted to the **rear**
    - "enqueue" to add iem to rear of queue
- Removal restricted to the **front**
    - "dequeue" to remove item from front of queue
- FIFO (first-in-first-out) structure
- Queue methods:
    - self.enqueue(item): inserts item at rear of queue
    - self.dequeue( ): removes and returns item at front of queue (queue must not be empty)
    - self.peak( ): returns item at front of queue
    - self.isEmpty( ): returns True if queue is empty, otherwise False
    - self.__len__( ): returns number of items in queue
    - self.__str__( ): returns string representation of queue
- Implementation:
    - Linked Structure:

```
class Node:
    def __init__(self, data, next):
        self.data = data
        self.next = next

class LinkedQueue( object ):
    def __init__(self):
        self.front = None
```

```
        self.size = 0
        self.rear = None

    def enqueue(self, newItem):
        newNode = Node( newItem, None )
        if self.isEmpty( ):
            self.front = newNode
        else:
            self.rear.next = newNode
        self.rear = newNode
        self.size += 1

    def dequeue(self):
        if self.isEmpty( ):
            print("Queue is empty")
            return ""
        else:
            oldItem = self.front.data
            self.front = self.front.next
            if self.front == None:          #queue is empty
                self.rear = None
            self.size -= 1
        return oldItem

    def peek(self):
        if self.isEmpty( ):
            print("Queue is empty")
            return ""
        else:
            return self.front.data

    def __len__(self):
        return self.size

    def isEmpty(self):
        return self.size == 0

    def __str__(self):
        res = ""
        curr = self.front
        while curr != None:
```

```
            res += str( curr.data ) + " "
            curr = curr.next
        return res
```

- Linear Array:

```
class ArrayQueue(object):
    CAPACITY = 100

    def __init__(self):
        self.items = list( )
        for i in range ArrayQueue.CAPACITY:
            self.items.append( None )
        self.rear = -1
        self.size = 0

    def enqueue(self, newItem):
        if self.isFull( ):
            print("Queue is full")
            return ""
        else:
            self.rear += 1
            self.size += 1
            self.items[self.rear] = newItem

    def dequeue(self):
        if self.isEmpty( ):
            print("Queue is empty")
            return ""
        else:
            oldItem = self.items[0]
            for i in range ( self.size-1 ):
                self.items[i] = self.items[i+1]
            self.rear -= 1
            self.size -= 1
            return oldItem

    def peek(self):
        if self.isEmpty( ):
            print("Queue is empty")
            return ""
```

```
        else:
            return self.items[0]

    def __len__(self):
        return self.size

    def isEmpty(self):
        return self.size == 0

    def isFull(self):
        return self.size == ArrayQueue.CAPACITY

    def __str__(self):
        res = ""
        if self.isEmpty( ):
            print("Queue is empty")
            return ""
        else:
            for i in range( self.size );
                res += str( self.items[i] ) + " "
            return res
```

- Cyclic Array:

```
class ArrayQueue:
    CAPACITY = 100

    def __init__(self):
        self.items = list( )
        for i in range ArrayQueue.CAPACITY:
            self.items.append(None)
        self.size = 0
        self.front = 0
        self.rear = -1

    def enqueue(self, newItem):
        if self.isFull( ):
            print("Queue is full")
            return ""
        else:
            if self.rear = ArrayQueue.CAPACITY - 1: #end of
```

```
array
                        self.rear = 0
                    else:
                        self.rear += 1
                    self.items[self.rear] = newItem
                    self.size += 1

            def dequeue(self):
                if self.isEmpty( ):
                    print("Queue is empty")
                    return ""
                else:
                    oldItem = self.items[self.front]
                    if self.front = ArrayQueue.CAPACITY - 1: #end
of array
                        self.front = 0
                    else:
                        self.front += 1
                    self.size -= 1
                    return oldItem

            def peek(self):
                if self.isEmpty( ):
                    print("Queue is empty")
                    return ""
                else:
                    return self.items[self.front]

            def __len__(self):
                return self.size

            def isEmpty(self):
                return self.size == 0

            def isFull(self):
                return self.size == ArrayQueue.CAPACITY

            def __str__(self):
                res = ""
                curr = self.front
                for i in range(self.size):
```

```
        res += str(self.items[curr]) + " "
        if curr = ArrayQueue.CAPACITY-1:
            curr = 0
        else:
            curr += 1
    return res
```