A2 Computer Science - Practicals

Complete Notes for Python & Pseudocode

Important Note

- When looking at the **space complexity** of algorithms, Cambridge **considers the <u>size of the structure</u> being used in the algorithm as a contributing factor** to its complexity - whereas usually, only auxiliary space (i.e. extra space) is considered.
- Hence, any algorithms that use **arrays** would have a space complexity of $O(n)$ or greater.

Reference: Cambridge International AS & A Level Computer Science - David Watson, Helen Williams - Page 490

Checklist

- [ ] Write algorithms for linear search and binary search
- [ ] Write algorithms for insertion sort and bubble sort
- [ ] Write algorithms for stacks (initialization, push, pop)
- [ ] Write algorithms for (linear and circular) queues (initialization, enqueue, dequeue)
- [ ] Write algorithms for (array-based) linked lists; (initialization, add, delete, search)
- [ ] Write algorithms for (array-based and OOP) binary trees; (initialization, add, search, traversal; pre-order, in-order and post-order)
- [ ] Write algorithms for initializing records
- [ ] Write algorithms for recursion
- [ ] Write algorithms using object-oriented programming (OOP)
    - [ ] Establish classes with a constructor, getters, setters and destructor
    - [ ] Establish constructors with/without parameters
    - [ ] Initialize public/private attributes
    - [ ] Establish inheritance (i.e. parent and child classes)
    - [ ] Establish polymorphism
        - [ ] Establish method overloading (i.e. methods within a class that have different signatures/prototypes)
        - [ ] Establish method overriding (i.e. methods within a child class that re-implements those of the parent class)
    - [ ] Instantiate objects of classes (including child classes) and arrays of objects
- [ ] Write algorithms using exception handling (i.e. `try-except` statements)
- [ ] Write algorithms using file handling
    - [ ] Read/write data to/from serial/sequential files
    - [ ] Read/write data to/from random files
    - [ ] Read/write data to/from files into/out of records (i.e. objects)
    - [ ] Read/write data to/from files with exception handling

Searching Algorithms

Linear Search

- **Brute force algorithm** to search for an item in a structure (e.g. array, linked list) by comparing it with *every element* in the structure.
- **Efficient linear search** must stop searching once the item is found.
- A variant of the following implementation could be to **return the index of where the item was found,** `-1` **otherwise**.

Time Complexity	Space Complexity
$O(n)$	$O(n)$

```python
items = [5, 4, 3, 6, 2, 7, 8, 1, 9, 0]  # items: ARRAY OF INTEGER

def linearSearch(item):  # item: INTEGER
    found = False  # found: BOOLEAN
    i = 0  # i: INTEGER

    while i < len(items) and found == False:
        if items[i] == item:
            found = True
        i += 1

    return found

print(linearSearch(5))  # Output: True
print(linearSearch(10))  # Output: False
```

Pseudocode

```
CONSTANT ARRAY_SIZE = 10

DECLARE items: ARRAY[1:ARRAY_SIZE] OF INTEGER

items <- [5, 4, 3, 6, 2, 7, 8, 1, 9, 0]

FUNCTION linearSearch(BYVAL item: INTEGER) RETURNS BOOLEAN
    DECLARE i: INTEGER
    DECLARE found: BOOLEAN

    found <- FALSE
    i <- 1

    WHILE i <= ARRAY_SIZE AND found = FALSE DO
        IF items[i] = item THEN
            found <- TRUE
        ENDIF
        i <- i + 1
    ENDWHILE

    RETURN found
ENDFUNCTION

OUTPUT linearSearch(5)  // Output: TRUE
OUTPUT linearSearch(10)  // Output: FALSE
```

Binary Search

- **Divide and conquer algorithm** to search for an item in a structure (e.g. array, heap) by checking the middle; otherwise repeating this process on the left or right halves of the structure; until an item is found or the respective half is empty.
- In order for this to work, the **structure must be sorted**; thereby adding overhead to the practical efficiency of Binary Search.
- A variant of the following implementation(s) could be to **return the index of where the item was found, `-1` otherwise**.
- There are two implementations of binary search; **iterative** and **recursive**.

Implementation	Time Complexity	Space Complexity
Iterative	$O(\log n)$	$O(n)$
Recursive	$O(\log n)$	$O(n \log n)$

Iterative Implementation

```python
items = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  # items: ARRAY OF INTEGER

def binarySearch(item):  # item: INTEGER
    found = False  # found: BOOLEAN
    left = 0  # left: INTEGER
    right = len(items)  # right: INTEGER

    while left <= right and found == False:
        mid = (left + right) // 2  # mid: INTEGER
        if items[mid] == item:
            found = True
        elif item > items[mid]:
            left = mid + 1
        else:
            right = mid - 1

    return found

print(binarySearch(5))  # Output: True
print(binarySearch(10))  # Output: False
```

Iterative Psuedocode Implementation

```
CONSTANT ARRAY_SIZE = 10

DECLARE items: ARRAY[1:ARRAY_SIZE] OF INTEGER

items <- [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

FUNCTION binarySearch(BYVAL item: INTEGER) RETURNS BOOLEAN
    DECLARE mid: INTEGER
    DECLARE left: INTEGER
    DECLARE right: INTEGER
    DECLARE found: BOOLEAN

    found <- FALSE
    left <- 1
    right <- ARRAY_SIZE

    WHILE left <= right AND found = FALSE DO
        mid <- (left + right) DIV 2
        IF items[mid] = item THEN
            found <- TRUE
        ELSE
            IF item > items[mid] THEN
                left <- mid + 1
            ELSE
                right <- mid - 1
            ENDIF
        ENDIF
    ENDWHILE

    RETURN found
ENDFUNCTION

OUTPUT binarySearch(5)  // Output: TRUE
OUTPUT binarySearch(10)  // Output: FALSE
```

Recursive Implementation

```python
items = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  # items: ARRAY OF INTEGER

def binarySearch(array, item):  # array: ARRAY OF INTEGER, item: INTEGER
    if len(array) == 0:
        return False

    m = len(array) // 2  # m: INTEGER
    if item == array[m]:
        return True
    elif item > array[m]:
        return binarySearch(array[m+1:], item)
    else:
        return binarySearch(array[:m], item)

print(binarySearch(items, 5))  # Output: True
print(binarySearch(items, 10))  # Output: False
```

Sorting Algorithms

Bubble Sort

- **Brute force algorithm** to sort a structure (i.e. array, linked list) in ascending/descending order of its elements by comparing and swapping every single element into its right position in the structure.
- **Efficient bubble sort** must stop sorting once *no more items are being swapped within a pass.*

Time Complexity	Space Complexity
$O(n^2)$	$O(n)$

- Maximum number of **passes**: $(n - 1)$
- Maximum number of **comparisons**: $\frac{n(n-1)}{2}$

```
items = [1, 5, 7, 6, 4, 3, 8, 9, 2, 0]  # items: ARRAY OF INTEGER

def bubbleSort():
    global items
    i = 0  # i: INTEGER
    swapped = True  # swapped: BOOLEAN
    # (1) outer loop for every "pass"
    #     ... if nothing was swapped by the end of a pass, the array is already sorted
    while i < len(array) - 1 and swapped:
        swapped = False
        # (2) inner loop for every "comparision"
        for j in range(len(array)-i-1):  # j: INTEGER
             # (3) check if the current item is greater than the next item
            if array[j] > array[j+1]:
                # (4) if true, swap the items
                array[j], array[j+1] = array[j+1], array[j]
                swapped = True
        i += 1

bubbleSort()
print(items)  # Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Insertion Sort

- Sorts a structure (i.e. array, linked list) in ascending/descending order of its elements by moving items from an *unsorted part* in the array to a *sorted part* (i.e. right to left)
- On average, **insertion sort** performs faster than **bubble sort** because when moving an item into the sorted part, it doesn't have to swap all its elements; just the ones larger than the item being inserted in. Hence, it can **stop the inner-loop early.**

Time Complexity	Space Complexity
$O(n^2)$	$O(n)$

```
items = [1, 5, 7, 6, 4, 3, 8, 9, 2, 0]  # items: ARRAY OF INTEGER

def insertionSort():
    global items
    # (1) outer loop for every "pass"
    for i in range(1, len(items)): # i: INTEGER
        value = items[i] # value: INTEGER
        j = i - 1 # j: INTEGER
        # (2) inner loop for moving the items in the sorted part to the right (i.e. making
space)
        while j >= 0 and value < items[j]:
            items[j + 1] = items[j]
            j -= 1
        # (3) insert item into the determined location in the sorted part
```

```
        items[j + 1] = value

insertionSort()
print(items)  # Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

User Defined Data Types

Records

- A **composite** data type comprising of *several related items that may be of different data types.*
- In Python, records are declared as **classes** with just a constructor *(type annotations must be included as comments)*

```
from datetime import date

class Student
    def __init__(self):
        self.name = ""  # STRING
        self.dob = date.today()  # DATE
        self.mark = 0  # INTEGER
```

- An **array of records** can be initialized as follows:

```
ARRAY_SIZE = 10  # CONSTANT
Students = [Student() for i in range(ARRAY_SIZE)]
```

- Following the initialization of the array, it can be **traversed and output using the following syntax**:

```
for student in Students:
    print(f"Student Name: {student.name} | Student DOB: {student.dob} | Student Mark:
{student.mark}")

# OUTPUT (assuming Students was given values):
# Student Name: Bob | Student DOB: 05-06-2003 | Student Mark: 89
# Student Name: Alice | Student DOB: 02-11-2003 | Student Mark: 75
# ...
```

Abstract Data Types (ADTs)

Stacks

- **LIFO (Last-in, First-out)** data structure (implemented as an array) that implements **push** to add an item to the top of the stack, and **pop** to remove an item from the top of the stack.
- A `topPointer` is required to save the **index of the item at top of the stack**
- A `basePointer` may be used to save the **index of the bottom of the stack** (i.e. lower bound of the array)
- `stackful` is a variable used to keep track of the **maximum size of the stack** (i.e. length of the array)
- An efficient **linear search** algorithm can be used to search a stack.

Initialization (Main Program)

```python
stackful = 5  # stackful: INTEGER
stack = [None for i in range(stackful)]  # stack: ARRAY OF INTEGER
topPointer = -1  # topPointer: INTEGER
basePointer = 0  # basePointer: INTEGER
```

`push()` Procedure

```python
def push(item):
    global stack, topPointer
    if topPointer == stackful - 1:
        print("The stack is full.")
    else:
        topPointer += 1
        stack[topPointer] = item
```

`pop()` Function

```python
def pop():
    global stack, topPointer
    item = None  # item: INTEGER
    if topPointer == basePointer - 1:
        print("The stack is empty.")
    else:
        item = stack[topPointer]
        stack[topPointer] = None
        topPointer -= 1
    return item
```

Example Main Program

```python
push(10)
push(15)
push(19)
push(14)

item = pop()
print(item)  # OUTPUT: 14

push(25)
push(15)
push(26)  # OUTPUT: The stack is full.

pop()
pop()
pop()
item = pop()
print(item)  # OUTPUT: 15

print(stack)  # OUTPUT: [10, None, None, None, None]
print("Top Pointer:", topPointer)  # OUTPUT: Top Pointer: 0
print("Base Pointer:", basePointer)  # OUTPUT: Base Pointer: 0
```

Linear Queue

- **FIFO (First-in, First-out)** data structure (implemented as an array) that implements **enqueue** to add an item to the rear of the queue, and **dequeue** to remove an item from the front of the queue.
- A `frontPointer` is required to save the **index of the item at the front of the queue** (used during *dequeue*)
- A `rearPointer` is required to save the **index of the item at the rear of the queue** (used during *enqueue*)
- `queueFul` is a variable used to keep track of the **maximum size of the queue** (i.e. length of the array)

Initialization

```
queueFul = 5  # queueFul: INTEGER
queue = [None for i in range(queueFul)]  # queue: ARRAY OF INTEGER
rearPointer = -1  # rearPointer: INTEGER
frontPointer = 0  # frontPointer: INTEGER
```

`enqueue()` Procedure

```
def enqueue(item):
    global queue, rearPointer
    if rearPointer == queueFul - 1:
        print("The queue is full.")
    else:
        rearPointer += 1
        queue[rearPointer] = item
```

`dequeue()` Function

```
def dequeue():
    global queue, frontPointer
    item = None  # item: INTEGER
    if frontPointer == queueFul:
        print("The queue is empty.")
    else:
        item = queue[frontPointer]
        queue[frontPointer] = None
        frontPointer += 1
    return item
```

- However the issue with a linear queue is that **when the `frontPointer` reaches the rear of the queue** (i.e. `queueFul`) - even when there's **space remaining at the front of the queue** - the locations at the front of the queue become unreachable.

Example Main Program

```
enqueue(10)
enqueue(15)
enqueue(19)
enqueue(14)

item = dequeue()
print(item)  # OUTPUT: 10

enqueue(25)
enqueue(15)  # OUTPUT: The queue is full.
```

```
enqueue(26)  # OUTPUT: The queue is full.

dequeue()
dequeue()
item = dequeue()
print(item)  # OUTPUT: 14

print(queue)  # OUTPUT: [None, None, None, None, 25]
print("Rear Pointer:", rearPointer)  # OUTPUT: Rear Pointer: 4
print("Front Pointer:", frontPointer)  # OUTPUT: Front Pointer: 4
```

Circular Queue

- An improved implementation of a linear queue with altered implementations for **enqueue** and **dequeue** - to allow for items to "loop back around" the rear of the queue to the front - eliminating unreachable locations.
- In addition to the previous variables, we'll also need `queueLength`; to save the **number of items currently in the queue**.

Initialization

```
queueFul = 5  # queueFul: INTEGER
queueLength = 0  # queueLength: INTEGER
queue = [None for i in range(queueFul)]  # queue: ARRAY OF INTEGER
rearPointer = -1  # rearPointer: INTEGER
frontPointer = 0  # frontPointer: INTEGER
```

`enqueue()` Procedure

```
def enqueue(item):
    global queue, queueLength, rearPointer
    # (1) check whether queue is full
    if queueLength == queueFul:
        print("The queue is full.")
    else:
        # (2) check whether the space is at the front / rear
        if rearPointer == queueFul - 1:
            rearPointer = 0
        else:
            rearPointer += 1
        # (3) add item to determined location
        queue[rearPointer] = item
        # (4) increment queue length
        queueLength += 1
```

dequeue() Function

```
def dequeue():
    global queue, queueLength, frontPointer
    item = None  # item: INTEGER
    # (1) check if queue is empty
    if queueLength == 0:
        print("The queue is empty.")
    else:
        # (2) remove the item at the front of the queue
        item = queue[frontPointer]
```

```
        queue[frontPointer] = None
        # (3) reset the front pointer to the beginning, or increment
        if frontPointer == queueFul:
            frontPointer = 0
        else:
            frontPointer += 1
        # (4) decrement queue length
        queueLength -= 1
    return item
```

Example Main Program

```
enqueue(10)
enqueue(15)
enqueue(19)
enqueue(14)

item = dequeue()
print(item)  # OUTPUT: 10

enqueue(25)
enqueue(15)
enqueue(26)  # OUTPUT: The queue is full.

dequeue()
dequeue()
item = dequeue()
print(item)  # OUTPUT: 14

print(queue)  # OUTPUT: [15, None, None, None, 25]
print("Rear Pointer:", rearPointer)  # OUTPUT: Rear Pointer: 0
print("Front Pointer:", frontPointer)  # OUTPUT: Front Pointer: 4
```

Linked List

- A **linear data structure** (implemented using an array) composed of **nodes** which each have a value and a pointer to the **next node**; until null.
- Besides the normal pointers in a linked list, it also has 2 *external pointers*:
  - `startPointer` : Index of the **first populated node in the linked list**.
  - `heapPointer / freePointer / freeList` : Index of the **next free node in the linked list.**
- A linked list has 4 major operations:
  1. `initialize()` : To reset and initialize all pointers and values, the `startPointer` and `heapPointer` of the linked list.
  2. `insert()` : To insert a value to the start of the linked list.
  3. `delete()` : To search and delete a value from the linked list.
  4. `find()` : To return whether or not an item was found in the linked list.
- The following implementation also has a **2 helper modules**:
  1. `findPrevious()` : To find the index of the node before the node you're search for; called within the `delete()` procedure to re-align the pointers.
  2. `tabulate()` : To display the linked list's indices, items and pointers in a tabular format.
- The above linked list operations have the following time complexities:

`insert()`	`delete()`	`find()`
$O(1)$	$O(n)$	$O(n)$

Declaration of Node Class + Variables

```
class Node:
    def __init__(self, item, pointer):
        self.item = item  # item: INTEGER
        self.pointer = pointer  # pointer: INTEGER

LIST_SIZE = 5  # CONSTANT
NULL_POINTER = -1  # CONSTANT

linkedList = [Node(0, NULL_POINTER) for i in range(LIST_SIZE)]  # linkedList: ARRAY OF Node
startPointer = NULL_POINTER  # startPointer: INTEGER
heapPointer = NULL_POINTER  # heapPointer: INTEGER
```

`tabulate()` Helper Procedure

```
def tabulate():
    print("INDEX\tITEM\tPOINTER")
    for i in range(LIST_SIZE): # i: INTEGER
        print(f"{i}\t{linkedList[i].item}\t{linkedList[i].pointer}")
    print("Start Pointer:", startPointer)
    print("Heap Pointer:", heapPointer)
```

`initialize()` Procedure

```
def initialize():
    global linkedList, startPointer, heapPointer
    # (1) set startPointer to null (since the list is initially empty)
    startPointer = NULL_POINTER
    # (2) set heapPointer to 0 (since the first free node is at index 0)
    heapPointer = 0
    # (3) connect all nodes to the next node
    for i in range(LIST_SIZE): # i: INTEGER
        linkedList[i].item = 0
        linkedList[i].pointer = i + 1
    # (4) set the last node's pointer to null (to indicate the end of the list)
    linkedList[LIST_SIZE - 1].pointer = NULL_POINTER
```

`insert()` Procedure

```
def insert(value):
    global linkedList, startPointer, heapPointer
    # (1) check if linked list is full
    if heapPointer == NULL_POINTER:
        print("Cannot insert, linked list is full.")
    else:
        # (2) otherwise, temporarily save the current free node's pointer
        tempPointer = linkedList[heapPointer].pointer  # tempPointer: INTEGER
        # (3) set the free node's item and pointer (to current start)
        linkedList[heapPointer].item = value
        linkedList[heapPointer].pointer = startPointer
        # (4) set startPointer to heapPointer (since the first node is now the free node)
        startPointer = heapPointer
```

```
        # (5) set the heapPointer to the temporarily saved pointer to the free node
        #     (set the heapPointer to the index of the node after the free node)
        heapPointer = tempPointer
```

`findPrevious()` Helper Function

```
def findPrevious(value):  # value: INTEGER
    # (1) establish pointers for the previous and current nodes
    prevPointer = NULL_POINTER - 1  # prevPointer: INTEGER
    currentPointer = startPointer  # currentPointer: INTEGER
    # (2) set found to False; to stop the search early
    found = False  # found: BOOLEAN
    # (3) search the linked list for a matching value until null is reached
    while currentPointer != NULL_POINTER and found == False:
        if linkedList[currentPointer].item == value:
            found = True
        else:
            prevPointer = currentPointer
            currentPointer = linkedList[currentPointer].pointer
    # (4) if not found, return null
    if found == False:
        prevPointer = NULL_POINTER
    # (5) otherwise, return prevPointer (i.e. index of the node before that found)
    return prevPointer
```

- The `findPrevious()` procedure returns one of **3 possible values:**
    - `-2` : represents that the item to delete is that at the **head** (i.e. start of the linked list)
    - `-1` : represents that the item to delete is **not found**.
    - `0 ... n` : item was **found**, and the index of the node previous to it is returned.

`delete()` Procedure

```
def delete(value):  # value: INTEGER
    global linkedList, startPointer, heapPointer
    # (1) determine the index of the node before that being deleted
    prevPointer = findPrevious(value)  # previousPointer: INTEGER
    # (2) check if the linked list is empty
    if startPointer == NULL_POINTER:
        print("Cannot delete, linked list is empty.")
    # (3) check if the prevPointer is (-1); item is not found
    elif prevPointer == NULL_POINTER:
        print("Cannot delete, item doesn't exist.")
    else:
        # (4) check if the prevPointer is (-2); item is found at the head
        currentPointer = startPointer  # currentPointer: INTEGER
        if prevPointer == NULL_POINTER - 1:
            startPointer = linkedList[currentPointer].pointer
        else:
            # (5) otherwise, item is found elsewhere
            currentPointer = linkedList[prevPointer].pointer
            linkedList[prevPointer].pointer = linkedList[currentPointer].pointer
        # (6) reset the deleted item's value and pointer
        linkedList[currentPointer].item = 0
        linkedList[currentPointer].pointer = NULL_POINTER
        # (7) set the heapPointer to accomodate for the deleted node's index
        heapPointer = currentPointer
```

- Based on the current implementation of `delete()`, **items cannot be inserted to the linked list after deletion** since the pointers are null'ed.
- Furthermore, any deleted nodes - although pointed to by the `heapPointer` - **become inaccessible thereafter.**
- In addition, you cannot `find()` items after `delete()`-ing them (since the connections to those nodes are lost)
- This can be prevented by using an additional array to keep track of those deleted locations, but is not implemented since it is **out of the syllabus' scope.**

`find()` Function

```
def find(value):  # value: INTEGER
    # (1) initialize pointer for the current node
    currentPointer = startPointer  # currentPointer: INTEGER
    # (2) set found to False; to stop the search early
    found = False  # found: BOOLEAN
    # (3) search the linked list for a matching value until null is reached
    while currentPointer != NULL_POINTER and found == False:
        if linkedList[currentPointer].item == value:
            found = True
        else:
            currentPointer = linkedList[currentPointer].pointer
    # (4) if found, return
    return found
```

Example Main Program

```
initialize()
tabulate()
# OUTPUT:
# INDEX   ITEM    POINTER
# 0       0       1
# 1       0       2
# 2       0       3
# 3       0       4
# 4       0       -1
# Start Pointer: -1
# Heap Pointer: 0

insert(50)
insert(20)
insert(10)
insert(80)
insert(70)
insert(40)  # OUTPUT: Cannot insert, linked list is full.
tabulate()
# OUTPUT:
# INDEX   ITEM    POINTER
# 0       50      -1
# 1       20      0
# 2       10      1
# 3       80      2
# 4       70      3
# Start Pointer: 4
# Heap Pointer: -1

print(find(20))  # OUTPUT: True
```

```
print(find(100))  # OUTPUT: False

delete(50)
delete(10)
delete(20)
delete(60)  # OUTPUT: Cannot delete, item doesn't exist.
tabulate()
# INDEX   ITEM    POINTER
# 0       0       -1
# 1       0       -1
# 2       0       -1
# 3       80      -1
# 4       70      3
# Start Pointer: 4
# Heap Pointer: 1
```

Binary Tree (Array-Based)

- A tree-structure that stored items in a **root, left branch and right branch** - in multiple levels. Items larger than the root are located in the right branch, while items smaller than the root is located in the left branch.
- Each item in a binary tree is a **node**; which has an item, a left pointer (i.e. reference to / index of the node on the left) and a right pointer (i.e. reference to / index of the node on the right)
- Besides the left and right pointers in a binary tree, it also has 2 *external pointers*:
    - `rootPointer` : Index of the **first populated node in the binary tree** (i.e. root node)
    - `freePointer` : Index of the **next free node in the binary tree**
- A binary tree has 3 major operations:
    1. `add()` : To insert a value to the binary tree.
    2. `find()` : To return whether or not an item was found in the binary tree.
    3. `traverse()` : To output the binary tree's items in one of 3 traversal orders; **pre-order, in-order and post-order**.
- The following implementation also has **1 helper module**:
    - `tabulate()` : To display the binary tree's indices, items, left pointers and right pointers in a tabular format.
- The above binary tree operations have the following time complexities:

`add()`	`find()`	`traverse()`
$O(1)$	$O(\log n)$	$O(n)$

- At the exam, the binary tree implementation can also be questioned as a **2D Array** - which is very similar to the implementation below, but instead of a `Node()` class, you'll be using a **3-element row to represent each node.**

Declaration of Node Class + Variables

```python
class Node:
    def __init__(self, item, leftPointer, rightPointer):
        self.item = item  # item: INTEGER
        self.leftPointer = leftPointer  # leftPointer: INTEGER
        self.rightPointer = rightPointer  # rightPointer: INTEGER

TREE_SIZE = 5  # CONSTANT
NULL_POINTER = -1  # CONSTANT

binaryTree = [Node(None, NULL_POINTER, NULL_POINTER) for i in range(TREE_SIZE)]  #
binaryTree: ARRAY OF Node
rootPointer = -1  # rootPointer: INTEGER
freePointer = 0  # freePointer: INTEGER
```

`tabulate()` Helper Procedure

```python
def tabulate():
    print("INDEX\tITEM\tLEFT\tRIGHT")
    for i in range(TREE_SIZE): # i: INTEGER
        print(f"
{i}\t{binaryTree[i].item}\t{binaryTree[i].leftPointer}\t{binaryTree[i].rightPointer}")
    print("Root Pointer:", rootPointer)
    print("Free Pointer:", freePointer)
```

`add()` Procedure

```python
def add(item):  # item: INTEGER
    global binaryTree, rootPointer, freePointer

    # (1) check if tree if empty; add to root
    if rootPointer == NULL_POINTER:
        rootPointer = 0
        binaryTree[rootPointer].item = item

    # (2) otherwise, see if free pointer has exceeded max. nodes
    elif freePointer >= TREE_SIZE:
        print("Cannot add, binary tree is full.")
        return

    # (3) otherwise, search where to add
    else:
        previousPointer = NULL_POINTER  # previousPointer: INTEGER
        currentPointer = rootPointer  # currentPointer: INTEGER
        addToLeft = False  # addToLeft: BOOLEAN
        # (4) until the current pointer is null...
        while currentPointer != NULL_POINTER:
            # ... (5) by first saving the parent node's index
            previousPointer = currentPointer
            # ... (6) traversing left branch if item less than root node's value
            if item < binaryTree[currentPointer].item:
                currentPointer = binaryTree[currentPointer].leftPointer
                addToLeft = True
            # ... (7) otherwise, right branch
            else:
                currentPointer = binaryTree[currentPointer].rightPointer
                addToLeft = False

        # (8) add the item to the new node (i.e. pointed to by the free pointer)
```

```
        binaryTree[freePointer].item = item

        # (9) connect the new node to the previous node
        if addToLeft:
            binaryTree[previousPointer].leftPointer = freePointer
        else:
            binaryTree[previousPointer].rightPointer = freePointer

    # (10) increment freePointer
    freePointer += 1
```

`find()` **Function** (Recursive)

```
def find(currentPointer, searchItem):
    # (1) return False if not found
    if currentPointer == NULL_POINTER:
        return False

    # (2) otherwise, check root; return True
    if binaryTree[currentPointer].item == searchItem:
        return True

    # (3) otherwise, traverse left branch (if smaller)
    elif searchItem < binaryTree[currentPointer].item:
        return find(binaryTree[currentPointer].leftPointer, searchItem)

    # (4) or, traverse right branch (if larger)
    else:
        return find(binaryTree[currentPointer].rightPointer, searchItem)
```

`traverse()` **Functions** - `preOrder()`, `inOrder()`, `postOrder()` (Recursive)

`preOrder()` **Procedure**

- Outputs the root node's item, then traverses the left branch, followed by the right branch.

  $Root \rightarrow Left \rightarrow Right$

```
def preOrder(currentPointer):
    # (1) return if currentPointer is null (i.e. -1)
    if currentPointer == NULL_POINTER:
        return

    # (2) output root node's item (in one-line)
    print(binaryTree[currentPointer].item, end=" ")

    # (3) fully traverse left branch (i.e. until null pointer is reached)
        preOrder(binaryTree[currentPointer].leftPointer)

    # (4) fully traverse right branch (i.e. until null pointer is reached)
    preOrder(binaryTree[currentPointer].rightPointer)
```

`inOrder()` **Procedure**

- Traverses the left branch, outputs the root node's item, and then traverses the right branch.

  $Left \rightarrow Root \rightarrow Right$

- The items will output **in ascending order**.

```
def inOrder(currentPointer):
    # (1) return if currentPointer is null (i.e. -1)
    if currentPointer == NULL_POINTER:
        return

    # (2) fully traverse left branch (i.e. until null pointer is reached)
    inOrder(binaryTree[currentPointer].leftPointer)

    # (3) output root node's item (in one-line)
        print(binaryTree[currentPointer].item, end=" ")

    # (4) fully traverse right branch (i.e. until null pointer is reached)
        inOrder(binaryTree[currentPointer].rightPointer)
```

`postOrder()` Procedure

- Traverses the left branch, the right branch and then outputs the root node's item

  $Left \rightarrow Right \rightarrow Root$

```
def postOrder(currentPointer):
    # (1) return if currentPointer is null (i.e. -1)
   if currentPointer == NULL_POINTER:
        return

    # (2) fully traverse left branch (i.e. until null pointer is reached)
    postOrder(binaryTree[currentPointer].leftPointer)

    # (3) fully traverse right branch (i.e. until null pointer is reached)
    postOrder(binaryTree[currentPointer].rightPointer)

        # (4) output root node's item (in one-line)
    print(binaryTree[currentPointer].item, end=" ")
```

Example Main Program

```
tabulate()
# OUTPUT:
# INDEX   ITEM    LEFT    RIGHT
# 0       None    -1      -1
# 1       None    -1      -1
# 2       None    -1      -1
# 3       None    -1      -1
# 4       None    -1      -1
# Root Pointer: -1
# Free Pointer: 0

add(50)
add(20)
add(80)
add(10)
add(70)
add(40)  # OUTPUT: Cannot add, binary tree is full.
tabulate()
# INDEX   ITEM    LEFT    RIGHT
# 0       50      1       2
# 1       20      3       -1
# 2       80      4       -1
# 3       10      -1      -1
```

```
# 4       70      -1      -1
# Root Pointer: 0
# Free Pointer: 5

print(find(rootPointer, 70))  # OUTPUT: True
print(find(rootPointer, 100))  # OUTPUT: False

preOrder(rootPointer)  # OUTPUT: 50 20 10 80 70
print("")  # to output newline after traversal

inOrder(rootPointer)  # OUTPUT: 10 20 50 70 80
print("")  # to output newline after traversal

postOrder(rootPointer)  # OUTPUT: 10 20 70 80 50
print("")  # to output newline after traversal
```

Binary Tree (OOP-based)

- A **dynamically re-sizable** implementation of the fixed-sized array-based Binary Tree, where the entire tree is built using a **Node class**, and all operations being methods of this class.
- This means that the `self` attribute can be now accessed, enabling **simpler recursive calls**.
- The `leftPointer` and `rightPointer` of the previous implementation is now replaced by `left` and `right` - which are **references to the left and right node objects; instead of indices.**
- The only external reference required in this implementation is a **root**; the object corresponding to the root node of the tree.
- All other operations are the same, and exhibit similar time complexities.

Declaration of Node Class + Methods

```
class Node:
    # (1) create the Node object's attributes
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

    # (2) add a new node to the tree
    def add(self, value):
        if value > self.value:
            if self.right == None:
                self.right = Node(value)
            else:
                self.right.add(value)
        else:
            if self.left == None:
                self.left = Node(value)
            else:
                self.left.add(value)

    # (3) find an item in the tree
    def find(self, item):
        if self.value == item:
            return True
        elif item > self.value:
            if self.right == None:
                return False
            else:
```

```
                return self.right.find(item)
        else:
            if self.left == None:
                return False
            else:
                return self.left.find(item)

    # (4) implement pre-order traversal
    def preOrder(self):
        # output root node's item
        print(self.value, end=" ")

        # fully traverse left branch (until null)
        if self.left != None:
            self.left.preOrder()

        # fully traverse right branch (until null)
        if self.right != None:
            self.right.preOrder()

    # (5) implement in-order traversal
    def inOrder(self):
        # fully traverse left branch (until null)
        if self.left != None:
            self.left.inOrder()

        # output root node's item
        print(self.value, end=" ")

        # fully traverse right branch (until null)
        if self.right != None:
            self.right.inOrder()

    # (6) implement post-order traversal
    def postOrder(self):
        # fully traverse left branch (until null)
        if self.left != None:
            self.left.postOrder()

        # fully traverse right branch (until null)
        if self.right != None:
            self.right.postOrder()

        # output root node's item
        print(self.value, end=" ")
```

Example Main Program

```
# establish root node + subsequent values
values = [50, 20, 80, 10, 70]
root = Node(values[0])
for value in values[1:]:
    # values[1:] is used to add all values except the first in the array
    root.add(value)

print(root.find(70))  # OUTPUT: True
print(root.find(100))  # OUTPUT: False

root.preOrder()  # OUTPUT: 50 20 10 80 70
```

```
print("")  # to output newline after traversal

root.inOrder()  # OUTPUT: 10 20 50 70 80
print("")  # to output newline after traversal

root.postOrder()  # OUTPUT: 10 20 70 80 50
print("")  # to output newline after traversal
```

Recursion

- Recursive algorithms are those **defined within a function, that calls itself.**
- The condition under which the function repeatedly calls itself is known as the **recursive case.** The opposite - the condition under which the function stops recursing and returns - is known as the **base case**.
- The process of calling a function in-on-itself until the base case is reached is known as **winding**, while the process of returning after reaching the base-case is known as **unwinding**.
- The above process is enabled by **maintaining a stack of function calls (i.e. call stack)** where each *call gets pushed upon recursion* and *popped upon return*.
- Recursion is used to write simple algorithms for **binary trees, binary search, factorials, counting, summation, checking for palindromes, merge sort and generating permutations** - here are 5 examples:

Example #1 - Determining the n^{th} factorial

```
def factorial(n):
 if n == 0:
     return 1
 else:
     return n * factorial(n - 1)
```

Example #2 - Determining S_n for the first n integers from 0.

```
def summation(n):
 if n == 0:
     return 0
 else:
     return n + summation(n - 1)
```

Example #3 - Determining whether input string s is a palindrome

- A **palindrome** is a word spelled the same way forwards and backwards; e.g. `radar`.

```
def is_palindrome(s):
    if len(s) <= 1:
        return True
    else:
        return s[0] == s[-1] and is_palindrome(s[1:-1])
```

Example #4 - Count the number of vowels in an input string s

```
def count_vowels(s):
 vowels = "aeiou"
 if not s:
     return 0
 else:
     return (1 if s[0].lower() in vowels else 0) + count_vowels(s[1:])
```

Example #5 - Convert a denary number into binary

```
def to_binary(n):
 if n == 0:
     return "0"
 elif n == 1:
     return "1"
 else:
     return to_binary(n // 2) + str(n % 2)
```

Object Oriented Programming (OOP)

Implementing Classes

- A class begins with a **constructor** (`__init__`) - which is the method that gets called when a class gets instantiated.
- All attributes of the class need to be declared and initialized within the constructor.
- The attributes can be set within the constructor in multiple different ways
  - **Default**: A default value is given to an attribute explicity (e.g. `IDNumber` is set to `0`)
  - **Parameterized**: A value is given to an attribute via a parameter defined in the constructor header (e.g. `FirstName`)
  - **Default Parameterized**: A value is given to an attribute via a parameter that also has a default value (e.g. `DOB` or `Grade`)
- Attributes can also be either **public** or **private** (defined with two leading underscores, e.g. `self.__IDNumber`) to implement **data hiding**.

```
class Student:
    def __init__(self, FirstName, DOB="01/01/1990", Grade=7):
        self.__IDNumber = 0  # PRIVATE IDNumber: INTEGER
        self.FirstName = FirstName  # FirstName: STRING
        self.DOB = DOB  # DOB: DATE
        self.Grade = Grade  # Grade: Grade
```

Setters and Getters

- Setters are methods that are **used to set values to attributes** of a class.
- Getters are method that are **used to return values of a class** to the main program.

```python
class Book:
    def __init__(self, ISBN, Title):
        # PRIVATE ISBN: STRING
        # DECLARE Title: STRING
        self.__ISBN = ISBN
        self.Title = Title
        self.BorrowerList = [None for i in range(5)]

    def getISBN(self):
        return self.__ISBN

    def setISBN(self, newISBN):
        self.__ISBN = newISBN
```

- In the above class, `getISBN` is used to **return the the current ISBN value** within a Book object.
- In contrary, `setISBN` is used to **update the existing ISBN value** with a `newISBN`.
- Getters and setters are most useful for **private attributes**; as they cannot be accessed from the main program explicitly:

```python
MyBook = Book("978-0735211292", "Atomic Habits")
print(MyBook.__ISBN)  # OUTPUT: AttributeError: 'Book' object has no attribute '__ISBN'
print(MyBook.getISBN())  # OUTPUT: 978-0735211292
```

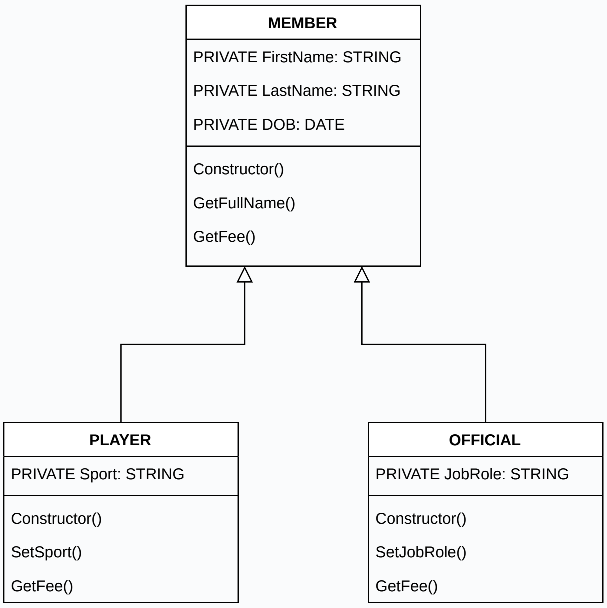
Destructors

- A class ends with a **destructor** (`__del__`) - which is the method that gets called when an object gets deleted.
- This is done automatically by Python's garbage collector; but explicitly call `del` in the exam.

```python
class Student:
    def __init__(self, FirstName, DOB="01/01/1990", Grade=7):
        # PRIVATE IDNumber: INTEGER
        # DECLARE FirstName: STRING
        # DECLARE DOB: DATE
        # DECLARE Grade: Grade
        self.__IDNumber = 0
        self.FirstName = FirstName
        self.DOB = DOB
        self.Grade = Grade

    def getIDNumber(self):
        return self.__IDNumber

    def setIDNumber(self, newIDNumber):
        self.__IDNumber = newIDNumber

    def __del__(self):
        print("Student was destroyed.")

MyStudent = Student("Edwin Fisher", "05/01/2003", 13)
del MyStudent  # OUTPUT: Student was destroyed.
```

Inheritance

- Inheritance is a *pillar of OOP* where a class **inherits attributes and methods** of a parent class.
- This way, the methods common to a class can be defined in the parent and those different can be defined in the child.
- When a child class is defined, the parent class **must also be initialized by either using `super()` or the parent class' name explicitly.**
- Consider the following **class diagram** (each box is a class; composed of a **class name, attribute list** and **method list**) for a sports club:



- In the above diagram, `PLAYER` and `OFFICIAL` **inherit** from the `MEMBER` parent class (**hierarchal inheritance**; one parent, multiple children).
- Let's imagine a scenario where each member is **charged a monthly fee of £50** to join the sports club. However, **players** are **deducted a percentage of this fee depending on the sport** they do:
  - `basketball`: 50% reduction
  - `badminton`: 20% reduction
  - any other sport: 10% reduction
- Officials of the sports club **don't have a fee.**
- This class diagram can be implemented is as follows:

```
# (1) establish parent class constructor, methods and destructor
class Member:
    def __init__(self, FirstName, LastName, DOB):
        # PRIVATE FirstName: STRING
        # PRIVATE LastName: STRING
        # PRIVATE DOB: DATE
        self.__FirstName = FirstName
        self.__LastName = LastName
        self.__DOB = DOB

    def GetFullName(self):
        return f"{self.__FirstName} {self.__LastName}"

    def GetFee(self):
```

```python
        return 50

    def __del__(self):
        print("Member was destroyed.")

# (2) establish the "Player" child class constructor, methods and destructor
class Player(Member):
    def __init__(self, FirstName, LastName, DOB):
        # (2.1) initialize the parent class
        super().__init__(self, FirstName, LastName, DOB)
        # ALTERNATIVE: Member.__init__(self, FirstName, LastName, DOB)
        self.__Sport = None  # Sport: STRING

    def SetSport(self, newSport):  # newSport: STRING
        self.__Sport = newSport

    def GetFee(self):
        if self.__Sport == "basketball":
            return super().GetFee() * 0.5
        elif self.__Sport == "badminton":
            return super().GetFee() * 0.8
        else:
            return super().GetFee() * 0.9

    def __del__(self):
        print("Player was destroyed.")

# (3) establish the "Official" child class constructor, methods and destructor
class Official(Member):
    def __init__(self, FirstName, LastName, DOB):
        # (3.1) initialize the parent class
        super().__init__(self, FirstName, LastName, DOB)
        # ALTERNATIVE: Member.__init__(self, FirstName, LastName, DOB)
        self.__JobRole = None  # JobRole: STRING

    def SetJobRole(self, newJobRole="Referee"):  # newJobRole: STRING
        self.__JobRole = newJobRole

    def GetFee(self):
        return 0

    def __del__(self):
        print("Official was destroyed.")
```

- In the above implementation, **polymorphism** aka. *overriding* (methods of the parent class is redefined for child classes) and **overloading** (method of a class being re-defined) can be seen:
    - **Polymorphism**: the `GetFee()` method of the `Member` parent class is **redefined for customized behavior** within the `Player` and `Official` child classes.
    - **Overloading**: The `SetJobRole()` method of the `Official` class has a **default parameter** of `"Referee"` for the parameter `newJobRole` - which means that this method can be called **with/without a `newJobRole` parameter**:
        - This is "technically" not proper overloading, but the best we can do with Python - since Python doesn't allow the same class to have two methods with the same name.
        - The other way of doing this is using an **`args` and `kwargs` parameter** - which is again out of the scope of the syllabus.

Instantiating Classes

- Instantiating a class is to **create objects using a defined class**.
- This object is in memory, and cannot be printed directly - it'll only print its reference in memory.
- You can access it's method and attributes using the **dot operator**; e.g. `MyStudent.Name`
- With reference to the previous Inheritance example, here's how the various classes can be instantiated:

```
# (1) creating an instance of the Player class
player1 = Player("John", "Doe", "1990-01-01")
player1.SetSport("basketball")

# (2) displaying the player's full name and fee
print(player1.GetFullName())  # OUTPUT: John Doe
print(f"Fee for basketball: {player1.GetFee()}")  # OUTPUT: 25.0 (50 * 0.5 due to discount)

# (3) creating an instance of the Official class with a default job role
official1 = Official("Alice", "Smith", "1985-03-15")
official1.SetJobRole()  # Default job role is set to 'Referee'

# (4) displaying the official's full name and fee
print(official1.GetFullName())  # OUTPUT: Alice Smith
print(f"Fee for official: {official1.GetFee()}")  # OUTPUT: 0 (Officials do not have a fee)

# (5) calling the destructors
del player1  # OUTPUT: "Player was destroyed."
del official1  # OUTPUT: "Official was destroyed."
```