1.**Searching Stack**

Write in pseudo code the following for the method on a Stack

def contains(stackName: Stack, search: str):

#input Stack stackName and searchString str

return bool

This will take a Stack of characters in a set sequence and a String and return True

if the Stack contains this string of characters in the same sequence.

Return false if not found in the Stack.

The elements in the Stack must return to their original order once this method is complete.

The Pseudo code for the contains method:

```
function contains(stackName: Stack, search: str) -> bool:
    # Create a temporary stack to hold the elements
    tempStack = createEmptyStack()
    # Initialize variables
    found = False
    searchIndex = 0

    # Traverse stackName & check for sequence
    while not isEmpty(stackName):
        # Pop character from stackName
```

```
        char = pop(stackName)

        # Save the character to tempStack
        push(tempStack, char)

        # Check if the character matches the search string
        if char == search[searchIndex]:
            # Move to the next character in the search string
            searchIndex += 1

            # Check if the entire search string has been matched
            if searchIndex == length(search):
                found = True
                break
        else:
            # If there's no match, reset searchIndex
            searchIndex = 0

    # Restore the original stack from tempStack
    while not isEmpty(tempStack):
```

push(stackName, pop(tempStack))

Return the result of the search

return found

Example:

Initial Stack : [2, 0, 1, 3, 6] (Top is 2)

Search String : 136

- Process Stack :
  - Pop 2, 0, push to tempStack : [2, 0] (not a match)
  - Pop 1, push to tempStack : [2, 0, 1] (matches search [0])
  - Pop 3, push to tempStack : [2, 0, 1, 3] (matches search[1])
  - Pop 6, push to tempStack : [2, 0, 1, 3, 6] (matches search[2])
- Restore Stack :
  - Pop 6, 3, 1, push back to original stack : [6, 3, 1]
  - Push back 0, 2 : [6,3,1,0,2]
- Return : True
- Final Stack : [2, 0, 6, 3, 1]

2. **Queue Class**

   Implement a Queue as a linked list in pseudocode, no full coding accepted for this one. Assume the data entry is hard coded, you do not enter from terminal. Your Linked List object must include at least the methods required to define the Abstract

Data Type (ADT) Queue. Write up a formula for the big O complexity of each algorithm in your program (using counting techniques).

Queue is an abstract data type (ADT) that manages a collection of elements in a first-in-first-out (FIFO) order, with references to the first (front) and last (rear) nodes, and performs functions such as Enqueue, Dequeue, Peek, IsEmpty, and Size operations on the queue.

- A node is a basic part of a linked list that holds a value (data) and a link to the next node in the sequence.
- The Enqueue(Data) function adds a new item to the end of the queue, whereas Dequeue() removes and returns the item at the front of the queue.
- Next, the peek() function displays the front item without removing it. Then the IsEmpty() function checks whether the queue is empty.
- Lastly, Size() counts the queue's items and returns that total.

Pseudo code for linked list queue:

Node:

Data // Holds the value

Next // Points to the next Node

Queue:

Front ← null // Points to the first Node

Rear ← null // Points to the last Node

Enqueue(Data):

Node ← Create a new Node

```
    Node.Data ← Data
    Node.Next ← null

    if Front is null:
        Front ← Node
        Rear ← Node
    else:
        Rear.Next ← Node
        Rear ← Node

Dequeue():
    if Front is null:
        return "Queue is empty"
    else:
        TempNode ← Front
        Front ← Front.Next
        if Front is null:
            Rear ← null
        return TempNode.Data
```

```
Peek():
    if Front is null:
        return "Queue is empty"
    else:
        return Front.Data


IsEmpty():
    return Front is null


Size():
    Count ← 0
    Current ← Front
    while Current is not null:
        Count ← Count + 1
        Current ← Current.Next
    return Count
```

Counting techniques involve basic operations such as assignments or loops, which are based on the number of elements in each queue method. Then, express the results as functions of n.

Big-O Complexity Formula:

- Enqueue: TEnqueue(n) = 6, O(1)
- Dequeue: TDequeue(n)=7 , O(1)
- Peek: TPeek(n)= 3 , O(1)
- IsEmpty: TIsEmpty(n)= 3, O(1)
- Size: TSize(n) = 2+3n+1 = 3n+3, O(n)

Queue operations (Enqueue, Dequeue, Peek, and IsEmpty) have a constant time complexity of O(1), while the size operation has a linear time complexity of O(n) because of the loop.

3.**Complexity**

Given the following code, what is its complexity for a list of length n. You need to show working but not a formula. Ignore commented lines with ”#”.

```
function nthFib(n) {
# let cache = {}; //Start here
function recurse(num) {
# if(cache[num]) :
# return cache[num] //second step
if(num === 0 || num == 1) return 1
let result = recurse(num-1) + recurse(num-2)
# cache[num] = result; //final step
return result;
}
return recurse(n);
}
```

What is the complexity if we uncomment the ’cache’ lines. Why?

How this code works:

- The function recurse(num) calls two times,such as recurse(num - 1) and recurse(num - 2).
- A binary tree develops quickly by each call generating two more calls.
- The count of calls nearly doubles at every degree of recursion. This produces a lot of function calls, particularly as n rises.
- Lack of caching causes the same Fibonacci numbers to be repeatedly computed.

Complexity for a list of length n

The nth Fib(n) function's temporal complexity for a list of length n is exponential. $O(2^n)$ is the time complexity as the number of function calls increases rapidly with n.

The complexity if uncomment the 'cache' lines:

- The function checks if `num` is in the cache. If found, it returns the cached result.

- If `num` is not cached, the function computes it by calling `recurse(num - 1)` and `recurse(num - 2)`.

- After computation, the result is stored in the cache for future use.

- **Time Complexity**: O(n) because each number is computed once.
- **Space Complexity:** O(n) due to the cache storing results for numbers up to n**.**

4. Heap

Give the below array of items, show all the steps in images that you would perform to create a max-heap

from this array, assuming the given array is the initial tree representation when you start the routine.

The change in array was needed as well as the tree heap, not full marks if miss one.

Also provide the pseudo/code for the sort routine.

Index	0	1	2	3	4	5	6	7	8
Array	2	7	4	13	9	10	5	6	13

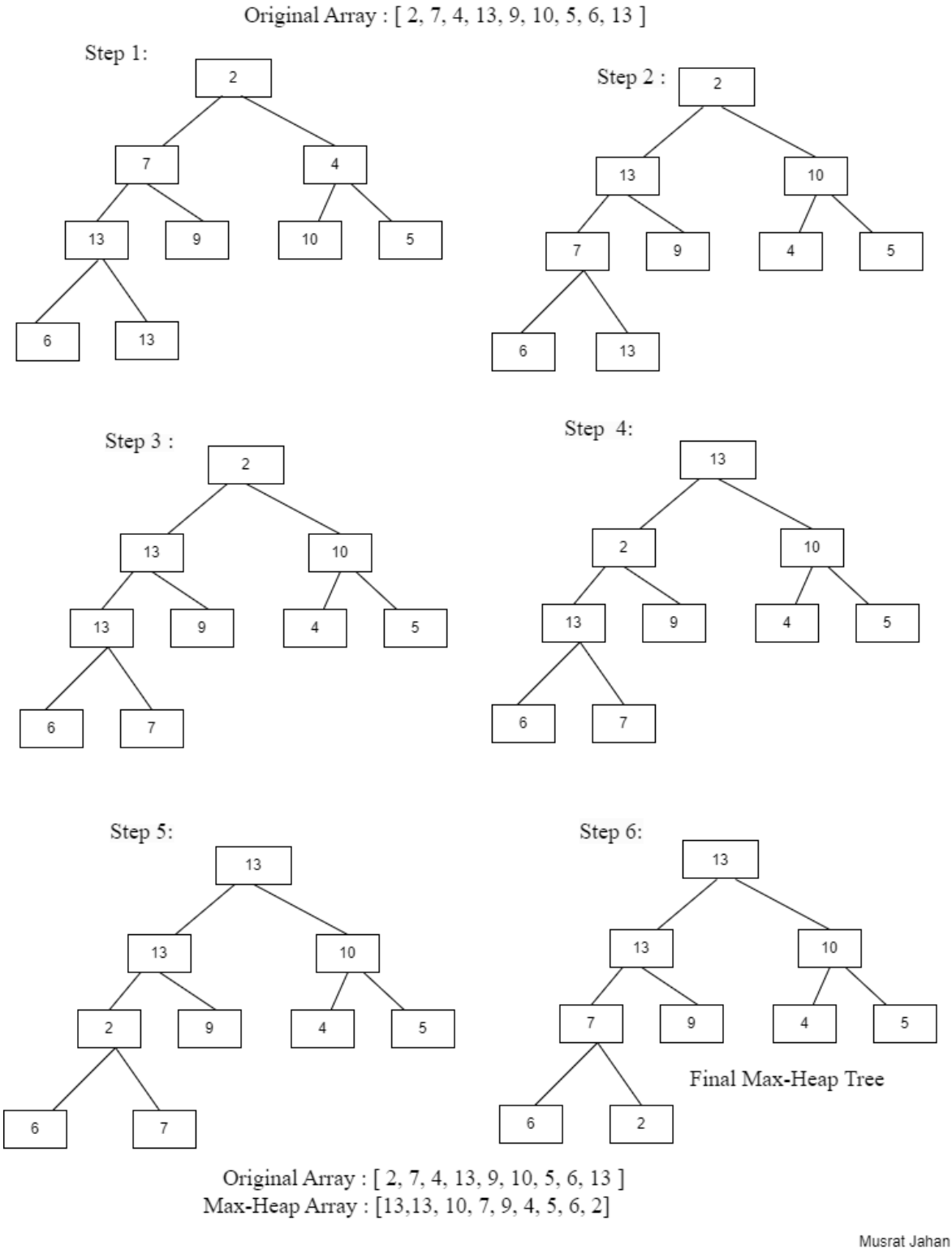
Table 1: Original Array

A heap is a complete binary tree where each node's key is greater than or equal to its parent's key, except for the root. It is often used to implement a priority queue, where elements are processed based on their priority. Using a heap allows for efficient insertion and extraction of the highest or lowest priority element in O(log n) time.

A max-heap is used when the highest priority element is the largest element, ensuring that the maximum element is always at the root and every parent node's key greater than or equal to its children's keys.

Array: [2,7,4,13,9,10,5,6,13]

Max – Heap Tree:



Original Array: [2, 7, 4, 13, 9, 10, 5, 6, 13]

Max-Heap Array: [13, 13, 10, 7, 9, 4, 5, 6, 2]

Pseudo code:

```
function maxHeapify(arr, n, i):
    largest = i            // Initialize largest as root
    left = 2 * i + 1       // left child index
    right = 2 * i + 2      // right child index

    // If left child exists and is greater than root
    if left < n and arr[left] > arr[largest]:
        largest = left

    // If right child exists and is greater than largest so far
    if right < n and arr[right] > arr[largest]:
        largest = right

    // If largest is not the root
    if largest != i:
        swap arr[i] with arr[largest] // Swap the root with the largest element
        maxHeapify(arr, n, largest)   // Recursively heapify the affected sub-tree

function buildMaxHeap(arr, n):
```

```
    // Start from the last non-leaf node and go up to the root node
    for i = (n // 2) - 1 down to 0:
        maxHeapify(arr, n, i)


function heapSort(arr):
    n = length(arr)
    buildMaxHeap(arr, n)


    // One by one extract elements from the heap
    for i = n - 1 down to 1:
        swap arr[0] with arr[i]       // Move current root to the end
        maxHeapify(arr, i, 0)         // Heapify the reduced heap
```

5. Storage

Consider any structures we have discussed: [Stack, Queue, Priority Queue, Array, Heap, Sorted/Unsorted List] You wish to create a database of clients for your shop. The most 10 recent clients you want to be shown at all times on your screen. However when a client returns you want them placed at the top of the screen. What data structure/s would you use. Briefly justify your choice. If you use more than one structure within another then use both, if you move from one to another then just show one that stores data for most of the time.

Data Structures: Deque (Double-ended Queue)

A deque allows insertion and deletion of elements from both the front and back. Add to the front of the deque. If the deque exceeds 10 clients, remove the least recent client from the back. A sorted list enables one to locate and change the position of a returning client so effectively placing returning clients at the top. Locate the client in the sorted list (or hash map). Remove the client from its current position in the deque. Add the client to the front of the deque.

Justification:

- Arrays have a fixed size and require rearranging elements to add or remove clients, making them inefficient.
- Stack Follows a LIFO order, thus clients that are not at the top cannot be easily accessed or reordered.
- Queue works for adding or removing clients in FIFO order, but does not effectively allow pushing returning customers to the front.
- Priority queue is designed for priority-based retrieval rather than recency.
- Sorted list is not essential to maintain a sorted order for recent consumers and reordering is wasteful.
- An unsorted list requires human administration to track timing and efficiently remove or transfer clients.

Deque is appropriate for managing the most recent 10 customers and encourage returning customers to the top, so facilitating quick adds and removals from both sides.