Problem 1

Part a:

Insertion Algorithm of Array

To insert element at the end of array, following steps are taken:

- Increase array size by 1.
- Assign the newly created index to the inserting item.

Pseudo Code:

```
procedure arrayAppend(A, newElement):
    A.length = A.length + 1
    A[A.length - 1] = newElement
end procedure
```

To insert element at any location in array, following steps are taken:

- Increase array size by 1.
- Move all elements, one index forward to make space for the new element.
- Insert the new element at the space created.

Pseudo Code:

```
procedure arrayInsert(A, newElement, position):
    for i from A.length - 1 to position + 1:
        A[i] = A[i - 1]
    A[position] = newElement
    A.length = A.length + 1
end procedure
```

Insertion Algorithm of Liked List

To Insert element at the start of Linked List:

- Make node for the new element.
- Point the *next* pointer of the created node to the *head* of linked list.
- Change the *head* of linked list to the newly created node.

Pseudo Code

```
procedure linkedListInsertBeginning(L, newElement):
    newNode = createNode(newElement)
    newNode.next = L.head
    L.head = newNode
end procedure
```

To insert element at any other location of Linked List:

- Make the node for new element.
- Move towards node of desired location, where the element needs to be inserted.
- Point the *next* pointer of the new node to the *next* of current node.
- Point the *next* of current node to the newly created node.

Pseudo Code

```
procedure linkedListInsert(L, newElement, position):
    newNode = createNode(newElement)
    current = L.head
    for i from 1 to position - 1:
        current = current.next
    newNode.next = current.next
    current.next = newNode
end procedure
```

Time Complexity of Array Insertion

Best Case: It is O(1), in which we are just appending the element ta the end.

Average Case: It is O(n), in which we need to shift elements. It takes linear time.

Worst Case: It is O(n), if we are inserting on the start, we would need to shift all elements by one.

Time Complexity of Linked List Insertion

Best Case: It is O(1). In this we are just inserting at the beginning which takes constant time.

Average Case: It is O(n). In this we need to traverse till insertion point.

Worst Case: It is O(n). If insertion point is at the end of the list, we need to traverse all.

Why Array should be preferred for Insertion

We can use array if:

- Fixed space is required.
- Random Insertion is required in code.
- Space Efficiency is required.

Why Linked List should be preferred for Insertion

We can use Linked List if:

- Frequent Insertion is required on start of list or at end.
- Dynamic size data structure is required.
- Memory efficiency is required for code.

Part b:

Search Algorithm for Array

If array is unsorted, it can be searched as follows:

Linear Search:

- Iterate through each element until the target element is found.
- Return -1, if target element not found in list.

Pseudo Code

```
function linearSearchUnsortedArray(A, target):
    for i from 0 to A.length - 1:
        if A[i] == target:
            return i
    return -1
end function
```

If array is sorted, we can apply binary search:

Binary Search:

1. Compare the target with the middle element of array.
2. If target element found then return the index, else:
3. If the target value is smaller than middle element, then search left part of array.
4. If the target value is larger than middle element, then search right part of array.
5. Repeat steps 2,3,4 until 1 element is left.
6. If target is found then return index, else return -1.

Pseudo Code

```
function binarySearchSortedArray(A, target):
    low = 0
    high = A.length - 1
    while low <= high:
        mid = (low + high) / 2
        if A[mid] == target:
            return mid
```

Search Algorithm for Linked List

If linked list is unsorted, it can be searched as follows:

Linear Search:

- Traverse the linked list from head.
- If target element found, then return the target node.
- Else return NULL pointer to node.

If linked list is sorted, it can be searched as follows:

Pseudo Code

```
function linearSearchUnsortedLinkedList(L, target):
    current = L.head
    while current is not null:
        if current.value == target:
            return current
        current = current.next
    return null
end function
```

Linear Search:

- Traverse the linked list from head.
- Traverse until the current node being checked is greater or equal to the target element.
- If target found at that time then return node, else return null.

Pseudo Code

```
function linearSearchSortedLinkedList(L, target):
    current = L.head
    while current is not null and current.value < target:
        if current.value == target:
            return current
        current = current.next
    return null
end function
```

Time Complexity

Time Complexity of Array Search:

- **Linear Search:** Its O(n), we need to traverse all elements of the unsorted array.
- **Binary Search:** Its O(logn), because in every iteration we divide the sorted array in half.

Time Complexity for Linked Lists:

- **Linear Search:** Its O(n) for both sorted and unsorted lists, because we need to traverse all elements in the list**.** In the worst case, whole sorted list is traversed.

Impact of sorting on time complexity

- **Array:** Sorting improves time complexity in array from O(n) to O(logn).
- **Linked List:** Sorting has no effect on time complexity of list.

Which data structure is preferred

Array will be preferred if:

- Frequent access to memory is required.
- Improve search complexity by applying binary search.

Linked Lists are preferred if:

- If insertion and deletion is less required as compared to insertion and deletion.

Part c:

Delete Algorithm for Array

To delete an element from array:

- Shift elements by 1 position towards the place where deleted element is placed.
- Decares the size of array by 1.

Pseudo Code

```
procedure arrayDelete(A, position):
    for i from position to A.length - 2:
        A[i] = A[i + 1]
    A.length = A.length - 1
end procedure
```

Delete Algorithm for Linked List

To delete an element from linked list:

- Traverse to the node before the node needs to be deleted.
- Point the *next* pointer of current to the *next pointer of next node.*

Pseudo Code

```
procedure linkedListDelete(L, position):
    if position == 0:
        L.head = L.head.next
    else:
        current = L.head
        for i from 1 to position - 1:
            current = current.next
        current.next = current.next.next
end procedure
```

Time Complexity

Time Complexity of Array Deletion:

- **At Beginning or Middle:** Its O(n), we need to traverse all elements of the array to reach the target.
- **At End:** Its O(1), because there is no need to shift.

Time Complexity for Linked Lists:

- **At Beginning:** Its O(1), since head needs to be updated only.
- **At end of middle:** Its O(n), because we need to traverse till target node.

Impact of sorting on time complexity

- **Array:** If target element is placed at end, its more effective.
- **Linked List:** Deletion at head is constant, whereas deletion further away takes more time.

Which data structure is preferred

Array will be preferred if:

- Frequent deletion is required at last element.

Linked Lists are preferred if:

- Deletion is required more at head of linked list.

Part d:

Memory Allocation when Data Structure is Initialized

Array

- This allocates blocks of memory equal to size of array at once.
- Blocks of memory are placed consecutively.

Linked List

- Memory for each node is allocated separately whenever required.
- More nodes are dynamically allocated.

Memory Allocation when Data Structure is at Half Capacity

Array

- Memory is already located for the size it was initialized.
- Half-filled array has no effect on memory allocation.

Linked List

- More memory is allocated when new nodes are added. It depends on the total number of nodes.

Memory Allocation when Data Structure is dynamically resized, or nodes are added/removed

Array

- In array memory resizing, new array of required size is allocated dynamically, and original contents of array are copied to it.
- This could lead to memory fragmentation if memory is not freed properly.

Linked List

- Adding or removing nodes involves allocating or deallocating memory for each node.
- Memory management is better than array in this.

Consideration / Recommendation

For choosing array

Arrays have efficient memory management if the size of dataset is known in advance, and it does not change. This efficiency decreases if arrays are frequently resized. It can be suitable for situations where memory usage is not concerned.

For choosing linked list

Linked lists have better memory management if the size of data set is not known. New nodes are dynamically added as new data is inserted into the linked list. It is suitable for conditions where size can be varied. If memory usage is critical in a program, then linked lists are preferred.

Problem 2

Part a:

Pseudo Code

```
class ArrayQueue:
    function initialize(capacity):
        self.capacity = capacity
        self.queue = new Array of size capacity
        self.front = -1
        self.rear = -1

    function is_empty():
        return front == -1

    function is_full():
        return (rear + 1) % capacity == front

    function enqueue(item):
        if is_full():
            print("Queue is full. Cannot enqueue.")
            return

        if is_empty():
            front = rear = 0
        else:
            rear = (rear + 1) % capacity

        queue[rear] = item
```

```
function dequeue():
    if is_empty():
        print("Queue is empty. Cannot dequeue.")
        return None

    removed_item = queue[front]

    if front == rear:
        front = rear = -1
    else:
        front = (front + 1) % capacity

    return removed_item

function peek():
    if is_empty():
        print("Queue is empty. Cannot peek.")
        return None
    return queue[front]
```

Limitations of Fixed Size

- The size of the queue is fixed while initializing the queue. Static array is used in it, because of which errors can occur if more elements are required to be put in array.
- The enqueing and dequeing functions may be required to implemented for circular array, which can increase complexity of the code.

Time Complexity

- **Enqueue:** It is O(1), because element is always added to the rear.
- **Dequeue:** It is O(1), because element is always removed from the front.
- **Peek:** It is O(1), because front element is accessed.

Part b:

```
class DynamicArrayQueue:
    function initialize(initial_capacity):
        self.capacity = initial_capacity
        self.queue = new Array of size initial_capacity
        self.front = -1
        self.rear = -1

    function is_empty():
        return front == -1

    function is_full():
        return (rear + 1) % capacity == front
    function enqueue(item):
        if is_full():
            resize()

        if is_empty():
            front = rear = 0
        else:
            rear = (rear + 1) % capacity

        queue[rear] = item
```

```
function resize():
    new_capacity = capacity * 2  # Double the capacity
    new_queue = new Array of size new_capacity

    # Copy existing elements to the new array
    index = 0
    current = front
    while current != -1:
        new_queue[index] = queue[current]
        current = (current + 1) % capacity
        index += 1

    front = 0
    rear = index - 1
    capacity = new_capacity
    queue = new_queue

function dequeue():
    if is_empty():
        print("Queue is empty. Cannot dequeue.")
        return None
    removed_item = queue[front]
    if front == rear:
        front = rear = -1
    else:
        front = (front + 1) % capacity
    return removed_item
```

Dynamic Resizing Strategy

- Resize() function is called to resize the queue, if its full.
- The front and rear indexes of the queue are adjusted accordingly in that function.
- Before enqueuing, it is checked if array has empty space or not.

Impact on Enqueue and Dequeue

- Resize function is called whenever queue is full, before enqueuing another element. It takes O(n) complexity as elements need to be copied to new array.
- Complexity for dequeue is same O(1), as dequeuing involves deleting an element at front.

Amortized Time Complexity

- The amortized time complexity of the enqueue operation is still O(1), as resizing is not very frequent, in this elements are copied from one array to new array, this results in average time complexity to be O(1).

```
class Node:
    function initialize(data):
        self.data = data
        self.next = null


class LinkedListQueue:
    function initialize():
        self.front = null
        self.rear = null


    function is_empty():
        return front is null


    function enqueue(item):
        new_node = createNode(item)
        if is_empty():
            front = rear = new_node
        else:
            rear.next = new_node
            rear = new_node
```

```
function dequeue():
    if is_empty():
        print("Queue is empty. Cannot dequeue.")
        return null

    removed_item = front.data
    if front == rear:
        front = rear = null
    else:
        front = front.next
    return removed_item

function peek():
    if is_empty():
        print("Queue is empty. Cannot peek.")
        return null
    return front.data
```

Part c:

Comparison with dynamic array-based queue

Linked List Queue

- Memory is allocated dynamically for each node in this queue. New node is added when required.
- There is no fixed memory, hence no need to resize. It makes memory more flexible.

Dynamic Array Based Queue

- Fixed amount of memory is allocated at initialization.
- Need to resize every time the array is full.

Performance

Linked List Queue

- Enqueuing and dequeuing operations are more efficient, as no resizing is required.
- Enqueuing and Dequeuing have O(1) complexity.

Dynamic Array Based Queue

- Resizing is required occasionally when array is full.
- Resizing has an impact on enqueue operation.
- It takes O(n) in worst case, to resize array. Due to this enqueue operation becomes O(n).

Time Complexity for Linked List Queue

- **Enqueue:** It is O(1), because element is always added to the rear.
- **Dequeue:** It is O(1), because element is always removed from the front.
- **Peek:** It is O(1), because front element is accessed.

Part d:

Array-Based Queue

1. *Dynamic Resizing with Overallocation*
   - Reduce the frequency of resizing processes by allocating more capacity than is required during resizing.
   - For instance, assign a new array with a capacity greater than the current number of entries when resizing.
2. *Implementing Circular Array*
   - To prevent elements from shifting during dequeue operations, use a circular array.
   - Instead of shifting elements when dequeuing, move the front pointer to the next index.
3. *Batch Enqueue and Dequeue*
   - Reduce the overhead of resizing and improve cache locality by using enqueue and dequeue operations.
   - For example, if it's feasible, enqueue or dequeue several elements simultaneously.

Linked List

1. *Node Pooling*
   - The overhead associated with dynamic memory allocation and deallocation can be greatly decreased by keeping a pool of pre-allocated nodes.
   - In order to reduce the frequency of dynamic memory operations, dequeued nodes are returned to the pool for possible reuse during subsequent enqueue operations.
2. *Memory efficient node Structures*
   - Optimizing the node structure itself can help reduce memory use inefficiencies.
   - If the data payload is small, think about utilizing a custom structure or a fixed-size array.
   - This lessens the memory overhead that each node requires.

3. *Bulk Memory Allocation*
    - Individual memory allocation can be avoided by allocating nodes in bulk instead of individually.
    - With this method, memory allocation calls are less frequent when the number of nodes required is known ahead of time.

Possible Trade-offs

1. *Memory Pooling Trade-off*
    - While memory pooling might increase productivity, it may also result in higher memory usage.
    - We can also consider trading-off between allocation speed and memory efficiency.
2. *Double Ended Queue Trade-off*
    - We can use double ended queue, but this can lead to more complex linked list implementation.
    - We can consider trading-off between functionality and simplicity.
3. *Node Catching Trade-off*
    - We can cache nodes to improve performance, but this can lead to increased memory usage.
    - We can consider trading-off between allocation speed and memory allocation.

Problem 3

Part a:

Breadth First Search

It is a graph traversal algorithm that searches the graph by levels. It starts from one source node, then explores all the neighbors of current breadth. After this it goes one level further and searches all the neighbors at that depth. BFS uses **queue** data structure to search all nodes layer by layer. Every time a node is explored, its neighbors are pushed into the queue. Then the element at the front is dequeued and the same process occurs with its neighbors.

Depth First Search

This is another graph searching tutorial that explores each node to its depth then moves to the other node. It goes as deep as possible then back tracks to explore the other branch. Due to this back tracking nature of DFS, **Stack** data structure is used to implement it. DFS is often used In problems involving cycles and path existence.

Which reaches the destination faster

Both DFS and BFS have their pros and cons. Both may reach their destination faster than the other depending on different situations.

When is BFS preferred

BFS is preferred while finding the shortest path because it explores all nodes of the current level before moving to the next depth. This leads to a fast shorter path finder algorithm.

Why is DFS preferred

In the same shorter path problem, DFS can work faster than BFS if the target node having the shortest path is placed deep in one of the branches. DFS searches an entire branch before to the next branch which is currently unexplored. Hence, in the same problem but different scenario DFS can work faster.

Part b:

BFS path to node 10

1. 1 will be visited. **Queue: 2,3.**
2. 2 will be visited. **Queue: 3,4,5.**
3. 3 will be visited. **Queue: 4,5,6,7.**
4. 4 will be visited. **Queue: 5,6,7,8,9.**
5. 5 will be visited. **Queue: 6,7,8,9,10,11.**
6. 6 will be visited. **Queue: 7,8,9,10,11,12,13.**
7. 7 will be visited. **Queue: 8,9,10,11,12,13,14,15**
8. 8 will be visited. **Queue: 9,10,11,12,13,14,15**
9. 9 will be visited. **Queue: 10,11,12,13,14,15**
10. 10 will be visited. **Queue: 11,12,13,14,15**
    Hence, 10 is visited and target node is found.

In this algorithm we are searching, level by level. We are searching for whole breadth then move to the next once. It took10 steps to reach the destination.

DFS path to 10

1. 1 will be visited. **Stack: 2,3.**
2. 2 will be visited. **Stack: 4,5,3.**
3. 4 will be visited. **Stack: 8,9,5,3.**
4. 8 will be visited. **Stack: 9,5,3.**
5. 9 will be visited. **Stack: 5,3.**
6. 5 will be visited. **Stack: 10,11,3.**
7. 10 will be visited. **Stack: 11,3.**
   Hence 10 is found and the target node is reached.

In this algorithm, we saw that we traverse towards the end of depth before moving to the next branch. Here DFS took less steps than BFS, but the difference is the if the target was on the right of source node. This would have greatly increased the number of steps taken by DFS in comparison to BFS.

Part c:

Convert 2021 into binary:

2021 = 11111100101

Now, reading the binary representation from left to right, follow these steps:

- Starting at vertex 1,
- Move Left when encountering a '0',
- Move Right when encountering a '1'.

The sequence of Left and Right moves for the path from vertex 1 to vertex 2021 is:

Right, Right, Right, Right, Right, Left, Left, Right, Left, Right

Justification:

- A '0' in the binary representation means move to the left child ($2k$).
- A '1' in the binary representation means move to the right child ($2k+1$).