Introductions to Arrays and LinkedList

I have gone through the videos and observed the difference between the Arrays and Linked List. Arrays are static in nature and Linked list are Dynamic nature. We can't increase the size of Arrays while run time. But it is possible in Linked list.

- Implemented the Creation, Deletion, Creation and Traverse methods for both Arrays and Linked List.

Creation, Insertion, Deletion and Traverse with arrays

Code:

```
package A1;
import java.util.*;
import java.util.Scanner;

public class Arr1
{
        static int[] arr;
        static int n ;

        public static void main(String[] args)
        {
                Scanner sc = new Scanner(System.in);

                while (true)
                 {
                        System.out.println("Array Operations Menu:");
                        System.out.println("1. Create Array");
                        System.out.println("2. Insert Element");
                        System.out.println("3. Delete Element");
                        System.out.println("4. Traverse Array");
                        System.out.println("5. Exit");
                        System.out.print("Choose an option: ");
                        int choice = sc.nextInt();

                        switch(choice)
                         {
                                case 1:
                                        arr = creation(sc);
```

```
                    break;
                case 2:
                    arr = insertion(arr,sc);
                    break;
                case 3:
                    arr = deletion(arr,sc);
                    break;
                case 4:
                    traverse();
                    break;
                case 5:
                    System.out.println("Exiting...");
                    sc.close();
                    return;
                default:
                    System.out.println("Invalid choice. Please choose a valid option.");
                    break;
            }
        }
    }

public static int[] creation(Scanner sc)
    {
        System.out.println("Enter the Size of the array : ");
        n = sc.nextInt();
        int[] arr = new int[n];
        System.out.println("Array created with size "+n);
        return arr;
    }

public static int[] insertion(int[] array, Scanner sc)
    {
        System.out.print("Enter index to insert (0 to " + (n - 1) + "): ");
        int index = sc.nextInt();
        if (index < 0 || index >= n)
        {
            System.out.println("Invalid index.");
            return array;
        }

        System.out.print("Enter the value to insert: ");
        int value = sc.nextInt();
        int[] arr = new int[array.length + 1];
        for (int i = 0; i < index; i++)
```

```
                {
                        arr[i] = array[i];
                }
                arr[index] = value;
                for (int i = index; i < array.length; i++)
                {
                        arr[i + 1] = array[i];
                }
                return arr;
        }

public static int[] deletion(int[] array, Scanner sc)
        {
                System.out.print("Enter index to delete (0 to " + (n - 1) + "): ");
                int index = sc.nextInt();
                if (index < 0 || index >= n)
                 {
                        System.out.println("Invalid index.");
                        return array;
                }
                System.out.println("Deleted element is "+array[index]+ " with position is
                "+index);
                int[] arr = new int[array.length - 1];
                for (int i = 0; i < index; i++)
                 {
                        arr[i] = array[i];
                }
                for (int i = index; i < arr.length; i++)
                 {
                        arr[i] = array[i + 1];
                }
                return arr;
        }

        public static void traverse()
        {
                if(n ==0)
                {
                        System.out.println("Array is not created. Please create an array first.");
                        return;
                }
                System.out.println("Array contents:");
                for (int i = 0; i < n; i++)
                {
                    System.out.println("Index " + i + ": " + arr[i]);
                    }
          }
}
```

Challenges faced with arrays

- Fixed size: cannot dynamically change size
- Insertion/Delection requires shifting elements, which is inefficient

Solutions:

- Dynamic arrays concept is used

Creation, Insertion, Deletion and Traverse with Linked List

Code:

```
package A1;
import java.util.Scanner;
public class linkedlist
{
static class Node
{
int data;
Node next;
Node(int data)
{
this.data = data;
this.next = null;
}
}
static class LinkedList
{
Node head;
void createList(int[] values) {
if(values.length ==0)
{
head = null;
return;
}
head = new Node(values[0]);
Node current = head;
for(int i=1;i<values.length; i++)
{
current.next = new Node(values[i]);
current = current.next;
}
}
void insertElement(int index, int value) {
Node newNode = new Node(value);
if (index == 0) {
newNode.next = head;
head = newNode;
return;
}
Node current = head;
for (int i = 0; i < index - 1; i++) {
if (current == null) {
System.out.println("Index out of bounds.");
return;
}
current = current.next;
}
newNode.next = current.next;
current.next = newNode;
}

void insertElement(int index, int value) {
Node newNode = new Node(value);
if (index == 0) {
newNode.next = head;
head = newNode;
return;
}
Node current = head;
for (int i = 0; i < index - 1; i++) {
if (current == null) {
System.out.println("Index out of bounds.");
return;
}
current = current.next;
}
newNode.next = current.next;
current.next = newNode;
```

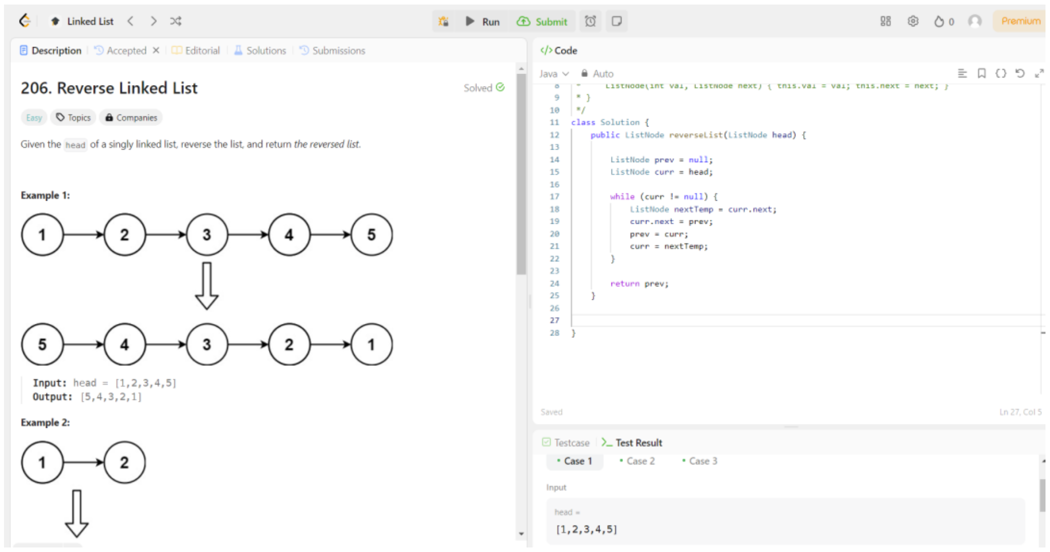
```
}
// Function to delete an element at a specific position
void deleteElement(int index) {
if (head == null) {
System.out.println("List is empty.");
return;
}
if (index == 0) {
head = head.next;
return;
}
Node current = head;
for (int i = 0; i < index - 1; i++) {
if (current == null || current.next == null) {
System.out.println("Index out of bounds.");
return;
}
current = current.next;
}
current.next = current.next.next;
}
void traverseList() {
Node current = head;
int index = 0;
while (current != null) {
System.out.println("Element at index " + index + ": " + current.data);
current = current.next;
index++;
}
}
}
public static void main(String[] args)
{
Scanner scanner = new Scanner(System.in);
LinkedList linkedList = new LinkedList();
while (true) {
System.out.println("Choose an operation:");
System.out.println("1. Create List");
System.out.println("2. Insert Element");
System.out.println("3. Delete Element");
System.out.println("4. Traverse List");
System.out.println("5. Exit");
int choice = scanner.nextInt();
switch (choice) {
case 1:
System.out.print("Enter the number of elements: ");
int n = scanner.nextInt();
int[] values = new int[n];
System.out.println("Enter the elements:");
for (int i = 0; i < n; i++) {
values[i] = scanner.nextInt();
}
linkedList.createList(values);
break;
case 2:
System.out.print("Enter the index to insert at: ");
int insertIndex = scanner.nextInt();
System.out.print("Enter the value to insert: ");
int insertValue = scanner.nextInt();
linkedList.insertElement(insertIndex, insertValue);
break;
case 3:
System.out.print("Enter the index to delete: ");
int deleteIndex = scanner.nextInt();
linkedList.deleteElement(deleteIndex);
break;
case 4:
linkedList.traverseList();
break;
case 5:
System.out.println("Exiting...");
scanner.close();
return;
default:
System.out.println("Invalid choice, please try again.");
}
}
}
}
```

Challenges with linked List

- Managing pointer is important
- For linked lists, you may need to traverse the entire list to find an element or insertion point.

Solutions:
- Ensure that all pointers are correctly assigned and released when nodes are inserted or deleted. Testing and debugging are critical to managing pointers correctly.
- Keep track of the tail pointer if you frequently need to access the end of the list.

I have done one example in Leet code



206. Reverse Linked List

Given the head of a singly linked list, reverse the list, and return *the reversed list*.

Example 1:

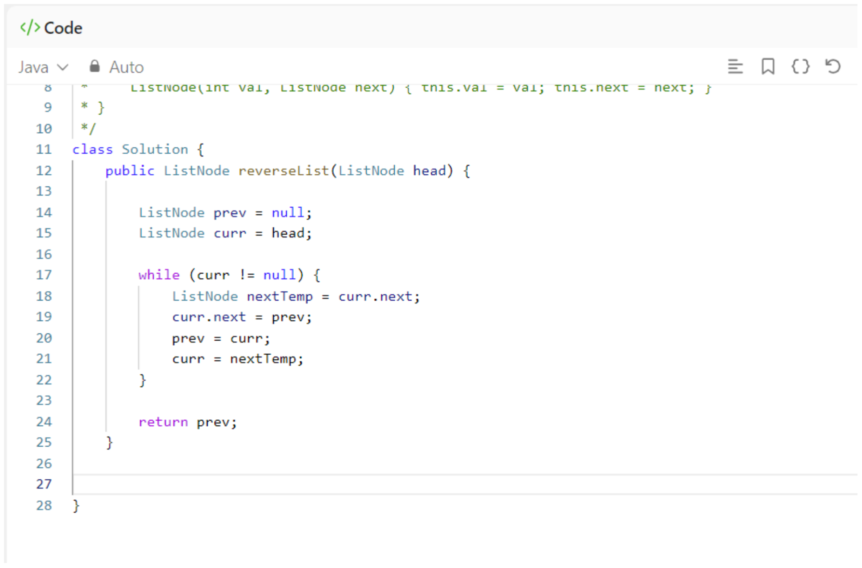
```
Input: head = [1,2,3,4,5]
Output: [5,4,3,2,1]
```

Example 2:

```
 * ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode reverseList(ListNode head) {

        ListNode prev = null;
        ListNode curr = head;

        while (curr != null) {
            ListNode nextTemp = curr.next;
            curr.next = prev;
            prev = curr;
            curr = nextTemp;
        }

        return prev;
    }

}
```

head =

[1,2,3,4,5]



```
 * ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode reverseList(ListNode head) {

        ListNode prev = null;
        ListNode curr = head;

        while (curr != null) {
            ListNode nextTemp = curr.next;
            curr.next = prev;
            prev = curr;
            curr = nextTemp;
        }

        return prev;
    }

}
```

Implementation:

Normally, In Single Linked List, don't have backward traverse. A node (A1) will store only a single address of other node (A2). But Node A2 will store the address of A3 or Null but it won't store address of A1. SO, we can implement reverse traverse in double link list because it contains the address of next node and previous node.

So, we can implement this reverse traverse by creating other pointers for storing the previous node address and we traverse in reverse direction.

```
package A1;
import java.util.Scanner;
public class Stackops {
static class Stack {
private int[] stackArray;
private int top;
private int capacity;
Stack(int size) {
capacity = size;
stackArray = new int[capacity];
top = -1;
}
void push(int value) {
if (isFull()) {
System.out.println("Stack is full. Cannot push " + value);
return;
}
stackArray[++top] = value;
System.out.println("Pushed " + value);
}
int pop() {
if (isEmpty()) {
System.out.println("Stack is empty. Cannot pop.");
return -1;
}
return stackArray[top--];
}
int peek() {
if (isEmpty()) {
System.out.println("Stack is empty. Cannot peek.");
return -1;
}
return stackArray[top];
}
boolean isEmpty() {
return top == -1;
}
boolean isFull() {
return top == capacity - 1;
}
void printStack() {
if (isEmpty()) {
System.out.println("Stack is empty.");
return;
```

```
}
System.out.println("Stack elements:");
for (int i = 0; i <= top; i++) {
System.out.println(stackArray[i]);
}
}
}
public static void main(String[] args) {
// TODO Auto-generated method stub
Scanner scanner = new Scanner(System.in);
System.out.print("Enter the size of the stack: ");
int size = scanner.nextInt();
Stack stack = new Stack(size);
while (true) {
System.out.println("Choose an operation:");
System.out.println("1. Push");
System.out.println("2. Pop");
System.out.println("3. Peek");
System.out.println("4. Print Stack");
System.out.println("5. Exit");
int choice = scanner.nextInt();
switch (choice) {
case 1:
System.out.print("Enter the value to push: ");
int pushValue = scanner.nextInt();
stack.push(pushValue);
break;
case 2:
int poppedValue = stack.pop();
if (poppedValue != -1) {
System.out.println("Popped " + poppedValue);
}
break;
case 3:
int peekValue = stack.peek();
if (peekValue != -1) {
System.out.println("Top element is " + peekValue);
}
break;
case 4:
stack.printStack();
break;
case 5:
System.out.println("Exiting...");
scanner.close();
return;
```

```
default:
System.out.println("Invalid choice, please try again.");
}
}
}
}
```

Queue

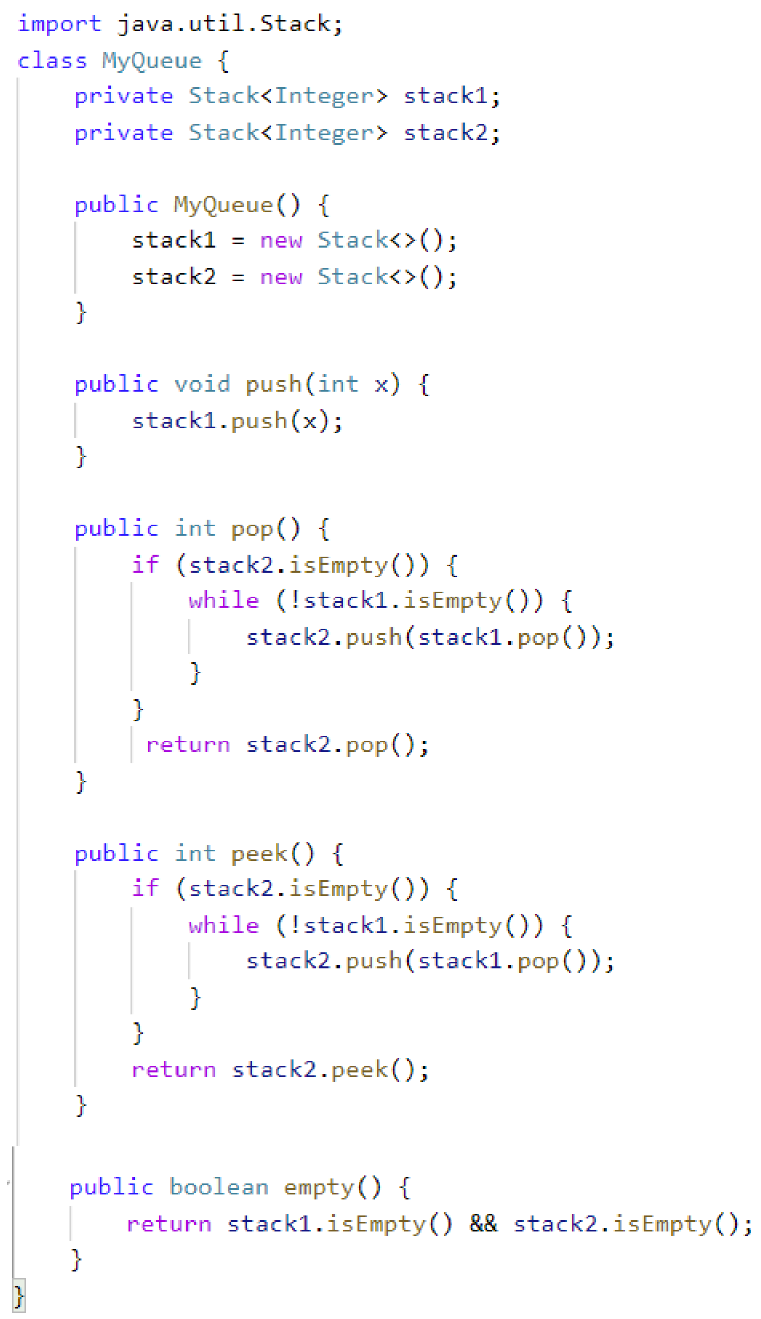
```
package A1;
import java.util.Scanner;
public class QueueOps {
private int[] queueArray;
private int front, rear, size, capacity;
QueueOps(int capacity) {
this.capacity = capacity;
queueArray = new int[capacity];
front = 0;
rear = -1;
size = 0;
}
void enqueue(int value) {
if (isFull()) {
System.out.println("Queue is full. Cannot enqueue " + value);
return;
}
rear = (rear + 1) % capacity;
queueArray[rear] = value;
size++;
System.out.println("Enqueued " + value);
}
int dequeue() {
if (isEmpty()) {
System.out.println("Queue is empty. Cannot dequeue.");
return -1;
}
int value = queueArray[front];
front = (front + 1) % capacity;
size--;
return value;
}
int peek() {
if (isEmpty()) {
System.out.println("Queue is empty. Cannot peek.");
```

```
return -1;
}
return queueArray[front];
}
boolean isEmpty() {
return size == 0;
}
boolean isFull() {
return size == capacity;
}
void printQueue() {
if (isEmpty()) {
System.out.println("Queue is empty.");
return;
}
System.out.println("Queue elements:");
for (int i = 0; i < size; i++) {
int index = (front + i) % capacity;
System.out.println(queueArray[index]);
}
}
//}
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.print("Enter the size of the queue: ");
int size = scanner.nextInt();
QueueOps queue = new QueueOps(size);
while (true) {
System.out.println("Choose an operation:");
System.out.println("1. Enqueue");
System.out.println("2. Dequeue");
System.out.println("3. Peek");
System.out.println("4. Print Queue");
System.out.println("5. Exit");
int choice = scanner.nextInt();
switch (choice) {
case 1:
System.out.print("Enter the value to enqueue: ");
int enqueueValue = scanner.nextInt();
queue.enqueue(enqueueValue);
break;
case 2:
int dequeuedValue = queue.dequeue();
if (dequeuedValue != -1) {
System.out.println("Dequeued " + dequeuedValue);
}
```

```
break;
case 3:
int peekValue = queue.peek();
if (peekValue != -1) {
System.out.println("Front element is " + peekValue);
}
break;
case 4:
queue.printQueue();
break;
case 5:
System.out.println("Exiting...");
scanner.close();
return;
default:
System.out.println("Invalid choice, please try again.");
}
}
}
}
```

Other Example:

Implementing Queue with Stack



```java
import java.util.Stack;
class MyQueue {
    private Stack<Integer> stack1;
    private Stack<Integer> stack2;

    public MyQueue() {
        stack1 = new Stack<>();
        stack2 = new Stack<>();
    }

    public void push(int x) {
        stack1.push(x);
    }

    public int pop() {
        if (stack2.isEmpty()) {
            while (!stack1.isEmpty()) {
                stack2.push(stack1.pop());
            }
        }
         return stack2.pop();
    }

    public int peek() {
        if (stack2.isEmpty()) {
            while (!stack1.isEmpty()) {
                stack2.push(stack1.pop());
            }
        }
        return stack2.peek();
    }

    public boolean empty() {
        return stack1.isEmpty() && stack2.isEmpty();
    }
}
```

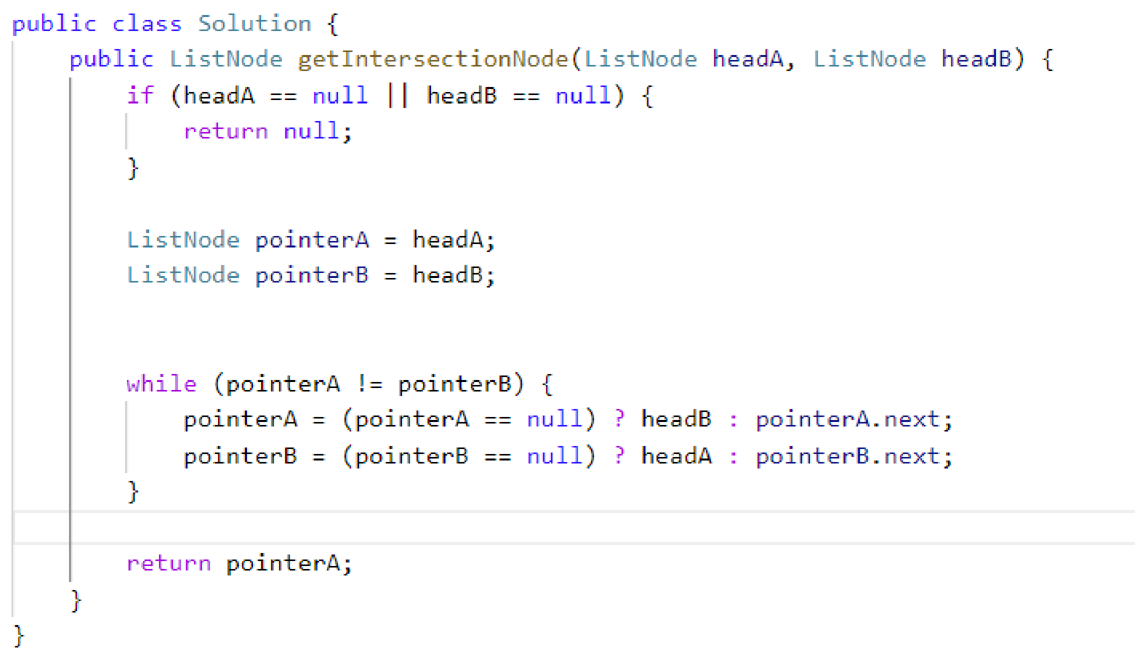
We know that a stack is LIFO and Queue is FIFO.. Is it possible to implement the Queue using stacks. Yes, we can implement by two stack. One stack is used for insertion and other stack is used for the reversing the stack 1 because LIFO. The element in stack 2 is complete reverse of stack 1. So, we do the pop operations in stack 2 to remove the first element first.

Hashtable

package A1;

```
import java.util.Hashtable;
import java.util.Scanner;
public class HashTableOps {
public static void main(String[] args) {
Hashtable<String, String> hashtable = new Hashtable<>();
Scanner scanner = new Scanner(System.in);
while (true) {
System.out.println("Hashtable Operations:");
System.out.println("1. Insert (Add a key-value pair)");
System.out.println("2. Delete (Remove a key-value pair)");
System.out.println("3. Search (Find a value by key)");
System.out.println("4. Update (Change a value by key)");
System.out.println("5. Exit");
System.out.print("Choose an option: ");
int choice = scanner.nextInt();
scanner.nextLine();
switch (choice) {
case 1:
System.out.print("Enter key: ");
String insertKey = scanner.nextLine();
System.out.print("Enter value: ");
String insertValue = scanner.nextLine();
hashtable.put(insertKey, insertValue);
System.out.println("Inserted: " + insertKey + " -> " + insertValue);
break;
case 2:
System.out.print("Enter key to delete: ");
String deleteKey = scanner.nextLine();
if (hashtable.remove(deleteKey) != null) {
System.out.println("Removed: " + deleteKey);
} else {
System.out.println("Key not found.");
}
break;
case 3:
System.out.print("Enter key to search: ");
String searchKey = scanner.nextLine();
String value = hashtable.get(searchKey);
if (value != null) {
System.out.println("Value for " + searchKey + " is " + value);
} else {
System.out.println("Key not found.");
}
break;
case 4:
System.out.print("Enter key to update: ");
String updateKey = scanner.nextLine();
System.out.print("Enter new value: ");
String updateValue = scanner.nextLine();
if (hashtable.containsKey(updateKey)) {
hashtable.put(updateKey, updateValue);
System.out.println("Updated: " + updateKey + " -> " + updateValue);
} else {
System.out.println("Key not found.");
}
break;
case 5:
System.out.println("Exiting...");
scanner.close();
return;
default:
System.out.println("Invalid choice. Please try again.");
}
}
}
}
```

Example of Hash Table



```
public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        if (headA == null || headB == null) {
            return null;
        }

        ListNode pointerA = headA;
        ListNode pointerB = headB;


        while (pointerA != pointerB) {
            pointerA = (pointerA == null) ? headB : pointerA.next;
            pointerB = (pointerB == null) ? headA : pointerB.next;
        }

        return pointerA;
    }
}
```

To finds the intersection node of two linked lists by using two pointers, pointerA and pointerB, initially set to the heads of each list. It traverses the lists, and when a pointer reaches the end of its list, it is redirected to the head of the other list. This ensures that both pointers cover the same total distance and thus meet at the intersection node if one exists. If there is no intersection, both pointers will eventually become null at the same time, and the method will return null.

Searching and Sorting

Bubblesort

```
package A1;
import java.util.Scanner;
public class Bubblesort {
public static void bubbleSort(int[] arr) {
int n = arr.length;
boolean swapped;
for (int i = 0; i < n - 1; i++) {
swapped = false;
for (int j = 0; j < n - i - 1; j++) {
if (arr[j] > arr[j + 1]) {
int temp = arr[j];
arr[j] = arr[j + 1];
arr[j + 1] = temp;
swapped = true;
}
}
if (!swapped) break;
}
}
public static void printArray(int[] arr) {
for (int num : arr) {
System.out.print(num + " ");
}
System.out.println();
}
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.print("Enter the number of elements: ");
int n = scanner.nextInt();
int[] arr = new int[n];
System.out.println("Enter " + n + " elements:");
for (int i = 0; i < n; i++) {
arr[i] = scanner.nextInt();
}
bubbleSort(arr);
System.out.println("Sorted array:");
printArray(arr);
scanner.close();
}
}
```

Challenges:

- **Time Complexity:** Bubble Sort has a time complexity of O(n^2), which makes it inefficient for large datasets.
- **Redundant Comparisons:** It performs unnecessary comparisons even after the array is sorted.

Solutions:

- **Optimization:** Implement an optimized version that stops early if no swaps are made in a pass.

Binary Search Algorithm

```
package A1;
import java.util.Scanner;
public class BinarySearch {
public static int binarySearch(int[] arr, int target) {
int left = 0;
int right = arr.length - 1;
while (left <= right) {
int mid = left + (right - left) / 2;
if (arr[mid] == target) {
return mid;
}
if (arr[mid] < target) {
left = mid + 1;
} else {
right = mid - 1;
}
}
return -1;
}
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.print("Enter the number of elements: ");
int n = scanner.nextInt();
int[] arr = new int[n];
System.out.println("Enter " + n + " sorted elements:");
for (int i = 0; i < n; i++) {
arr[i] = scanner.nextInt();
}
System.out.print("Enter the element to search for: ");
int target = scanner.nextInt();
int result = binarySearch(arr, target);
if (result == -1) {
System.out.println("Element not found.");
} else {
```

```
System.out.println("Element found at index: " + result);
}
scanner.close();
}
}
```

Challenges:
- **Requires Sorted Array:** Binary Search only works on a sorted array.
- **Index Handling:** Incorrect handling of indices can lead to errors or inefficiencies.

Solutions:
- **Pre-Sort Data:** Ensure the data is sorted before applying binary search.
- **Handle Edge Cases:** Carefully manage index calculations to avoid overflow or off-by-one errors.

BFS Algorithm

```
package A1;
import java.util.*;
public class BFS {
private int vertices;
private LinkedList<Integer> adjList[];
public BFS(int v) {
vertices = v;
adjList = new LinkedList[v];
for (int i = 0; i < v; i++) {
adjList[i] = new LinkedList<>();
}
}
public void addEdge(int source, int destination) {
adjList[source].add(destination);
}
public void BFS(int startVertex) {
boolean visited[] = new boolean[vertices];
LinkedList<Integer> queue = new LinkedList<>();
visited[startVertex] = true;
queue.add(startVertex);
while (queue.size() != 0) {
startVertex = queue.poll();
System.out.print(startVertex + " ");
Iterator<Integer> i = adjList[startVertex].listIterator();
while (i.hasNext()) {
int nextVertex = i.next();
if (!visited[nextVertex]) {
visited[nextVertex] = true;
queue.add(nextVertex);
}
}
}
}
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.print("Enter number of vertices: ");
int v = scanner.nextInt();
BFS graph = new BFS(v);
System.out.print("Enter number of edges: ");
int e = scanner.nextInt();
System.out.println("Enter the edges (source destination):");
for (int i = 0; i < e; i++) {
int source = scanner.nextInt();
int destination = scanner.nextInt();
graph.addEdge(source, destination);
}
System.out.print("Enter the starting vertex for BFS: ");
int startVertex = scanner.nextInt();
System.out.println("BFS traversal starting from vertex " + startVertex + ":");
graph.BFS(startVertex);
}
}
```

Challenges:

- **Memory Usage:** BFS uses a queue to keep track of nodes, which can consume a lot of memory for large graphs.
- **Time Complexity:** Time complexity is O(V + E) where V is the number of vertices and E is the number of edges, which can be high for dense graphs.

Solutions:

- **Optimize Queue Usage:** Use a queue implementation that efficiently handles node storage and retrieval.
- **Space Complexity Management:** Implement algorithms that only store nodes that are currently being explored.

DFS Algorithm

```
package A1;
import java.util.*;
public class DFS {
private int vertices;
private LinkedList<Integer> adjList[];
public DFS(int v) {
```

```
vertices = v;
adjList = new LinkedList[v];
for (int i = 0; i < v; i++) {
adjList[i] = new LinkedList<>();
}
}
public void addEdge(int source, int destination) {
adjList[source].add(destination);
adjList[destination].add(source); // For an undirected graph
}
public void DFS(int startVertex) {
boolean visited[] = new boolean[vertices];
DFSUtil(startVertex, visited);
}
private void DFSUtil(int vertex, boolean[] visited) {
visited[vertex] = true;
System.out.print(vertex + " ");
Iterator<Integer> i = adjList[vertex].listIterator();
while (i.hasNext()) {
int nextVertex = i.next();
if (!visited[nextVertex]) {
DFSUtil(nextVertex, visited);
}
}
}
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.print("Enter number of vertices: ");
int v = scanner.nextInt();
DFS graph = new DFS(v);
System.out.print("Enter number of edges: ");
int e = scanner.nextInt();
System.out.println("Enter the edges (source destination):");
for (int i = 0; i < e; i++) {
int source = scanner.nextInt();
int destination = scanner.nextInt();
graph.addEdge(source, destination);
}
System.out.print("Enter the starting vertex for DFS: ");
int startVertex = scanner.nextInt();
System.out.println("DFS traversal starting from vertex " + startVertex + ":");
graph.DFS(startVertex);
}
}
```

Challenges:

- **Stack Overflow:** DFS can use a lot of stack space in case of deep recursion, leading to stack overflow errors.
- **Finding Paths:** Can be inefficient if you need to find the shortest path.

Solutions:

- **Iterative Approach:** Use an iterative version with an explicit stack to avoid stack overflow.
- **Path Optimization:** Combine DFS with other techniques (like BFS) if you need to find the shortest path.

Linear Search algorithm

```
package A1;
import java.util.Scanner;
public class LinearSearch {
public static int linearSearch(int[] array, int target) {
for (int i = 0; i < array.length; i++) {
if (array[i] == target) {
return i;
}
}
return -1;
}
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.print("Enter the number of elements: ");
int n = scanner.nextInt();
int[] numbers = new int[n];
System.out.println("Enter the elements:");
for (int i = 0; i < n; i++) {
numbers[i] = scanner.nextInt();
}
System.out.print("Enter the target value: ");
int target = scanner.nextInt();
int result = linearSearch(numbers, target);
if (result != -1) {
System.out.println("Element found at index: " + result);
} else {
System.out.println("Element not found in the array.");
}
scanner.close();
}
}
```

Challenges:

- **Inefficiency:** Time complexity is O(n), making it inefficient for large datasets compared to more advanced search algorithms.

Solutions:

- **Pre-Sort Data:** If searching frequently, consider sorting the data and using binary search.
- **Optimized Linear Search:** For some cases, ensure that the search operation is as efficient as possible by avoiding redundant checks.

Bucket sort

```
package A1;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Scanner;
public class BucketSort {
public static void bucketSort(int[] array) {
if (array.length <= 1) return;
int max = array[0];
int min = array[0];
for (int num : array) {
if (num > max) max = num;
if (num < min) min = num;
}
int bucketCount = max - min + 1;
ArrayList<Integer>[] buckets = new ArrayList[bucketCount];
for (int i = 0; i < bucketCount; i++) {
buckets[i] = new ArrayList<>();
}
for (int num : array) {
buckets[num - min].add(num);
}
int index = 0;
for (ArrayList<Integer> bucket : buckets) {
Collections.sort(bucket);
for (int num : bucket) {
array[index++] = num;
}
}
}
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.print("Enter the number of elements: ");
int n = scanner.nextInt();
int[] numbers = new int[n];
System.out.println("Enter the elements:");
for (int i = 0; i < n; i++) {
numbers[i] = scanner.nextInt();
}
bucketSort(numbers);
System.out.println("Sorted array:");
for (int num : numbers) {
System.out.print(num + " ");
}
scanner.close();
}
}
```

Challenges:

- **Assumptions:** Requires assumptions about the distribution of the input data.
- **Space Complexity:** Can be space-intensive as it requires additional storage for buckets.

Solutions:

- **Distribution Management:** Use when the input is uniformly distributed and within a known range.
- **Memory Optimization:** Choose the number of buckets wisely to balance space and sorting time.

Couting sort Algorithm

```
package A1;
import java.util.Scanner;
public class CountingSort {
public static void countingSort(int[] array) {
if (array.length == 0) return;
int max = array[0];
int min = array[0];
for (int num : array) {
if (num > max) max = num;
if (num < min) min = num;
}
int range = max - min + 1;
int[] count = new int[range];
for (int num : array) {
count[num - min]++;
}
int index = 0;
for (int i = 0; i < count.length; i++) {
while (count[i]-- > 0) {
array[index++] = i + min;
}
}
}
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.print("Enter the number of elements: ");
int n = scanner.nextInt();
int[] numbers = new int[n];
System.out.println("Enter the elements:");
for (int i = 0; i < n; i++) {
numbers[i] = scanner.nextInt();
}
countingSort(numbers);
System.out.println("Sorted array:");
for (int num : numbers) {
System.out.print(num + " ");
}
scanner.close();
}
}
```

Challenges:

- **Space Complexity:** Space complexity is O(k) where k is the range of input values.
- **Range Limitation:** Effective only for integer keys with a small range.

Solutions:

- **Adjust Range:** Normalize or adjust the range of input values if necessary.
- **Alternative Implementations:** Consider using other sorting algorithms if the range of values is very large.

Radix sort Algorithm

```
package A1;
import java.util.Arrays;
import java.util.Scanner;
public class RadixSort {
private static void countingSort(int[] array, int exp) {
int n = array.length;
int[] output = new int[n];
int[] count = new int[10];
Arrays.fill(count, 0);
for (int i = 0; i < n; i++) {
count[(array[i] / exp) % 10]++;
}
for (int i = 1; i < 10; i++) {
count[i] += count[i - 1];
}
for (int i = n - 1; i >= 0; i--) {
```

```
output[count[(array[i] / exp) % 10] - 1] = array[i];
count[(array[i] / exp) % 10]--;
}
System.arraycopy(output, 0, array, 0, n);
}
private static void radixSort(int[] array) {
int max = Arrays.stream(array).max().orElse(0);
for (int exp = 1; max / exp > 0; exp *= 10) {
countingSort(array, exp);
}
}
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.print("Enter the number of elements: ");
int n = scanner.nextInt();
int[] numbers = new int[n];
System.out.println("Enter the elements:");
for (int i = 0; i < n; i++) {
numbers[i] = scanner.nextInt();
}
radixSort(numbers);
System.out.println("Sorted array:");
for (int num : numbers) {
System.out.print(num + " ");
}
scanner.close();
}
}
```

Challenges:

- **Space Complexity:** Uses additional space proportional to the number of digits and the range of input values.
- **Performance:** Can be inefficient for very large numbers or if the input data is not uniformly distributed.

Solutions:

- **Handle Large Data:** Optimize based on the number of digits and use efficient counting techniques.
- **Preprocessing:** Preprocess the data to fit the requirements of Radix Sort.

Heap Sort

```
package A1;
import java.util.Scanner;
public class HeapSort {
private static void heapify(int[] array, int n, int i) {
```

```
int largest = i;
int left = 2 * i + 1;
int right = 2 * i + 2;
if (left < n && array[left] > array[largest]) {
largest = left;
}
if (right < n && array[right] > array[largest]) {
largest = right;
}
if (largest != i) {
int swap = array[i];
array[i] = array[largest];
array[largest] = swap;
heapify(array, n, largest);
}
}
private static void heapSort(int[] array) {
int n = array.length;
for (int i = n / 2 - 1; i >= 0; i--) {
heapify(array, n, i);
}
for (int i = n - 1; i >= 0; i--) {
int temp = array[0];
array[0] = array[i];
array[i] = temp;
heapify(array, i, 0);
}
}
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.print("Enter the number of elements: ");
int n = scanner.nextInt();
int[] numbers = new int[n];
System.out.println("Enter the elements:");
for (int i = 0; i < n; i++) {
numbers[i] = scanner.nextInt();
}
heapSort(numbers);
System.out.println("Sorted array:");
for (int num : numbers) {
System.out.print(num + " ");
}
scanner.close();
}
}
```

Challenges:

- **Complexity:** Heap operations (insert and remove) can be complex to implement and understand.
- **Stability:** Heap Sort is not a stable sort.

Solutions:

- **Use Built-in Functions:** Many languages provide built-in functions for heap operations, which can be used to simplify implementation.
- **Understand Heap Operations:** Thoroughly understand heap operations (heapify, extract max/min) to avoid bugs.

Merge Sort

```
package A1;
import java.util.Scanner;
public class MergeSort {
private static void merge(int[] array, int left, int mid, int right) {
int n1 = mid - left + 1;
int n2 = right - mid;
int[] L = new int[n1];
int[] R = new int[n2];
System.arraycopy(array, left, L, 0, n1);
System.arraycopy(array, mid + 1, R, 0, n2);
int i = 0, j = 0;
int k = left;
while (i < n1 && j < n2) {
if (L[i] <= R[j]) {
array[k++] = L[i++];
} else {
array[k++] = R[j++];
}
}
while (i < n1) {
array[k++] = L[i++];
}
while (j < n2) {
array[k++] = R[j++];
}
}
private static void mergeSort(int[] array, int left, int right) {
if (left < right) {
int mid = (left + right) / 2;
mergeSort(array, left, mid);
mergeSort(array, mid + 1, right);
merge(array, left, mid, right);
}
}
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.print("Enter the number of elements: ");
int n = scanner.nextInt();
int[] numbers = new int[n];
System.out.println("Enter the elements:");
for (int i = 0; i < n; i++) {
numbers[i] = scanner.nextInt();
}
mergeSort(numbers, 0, n - 1);
System.out.println("Sorted array:");
for (int num : numbers) {
System.out.print(num + " ");
}
scanner.close();
}
}
```

Challenges:

- **Space Complexity:** Merge Sort requires additional space proportional to the array size.
- **Complex Implementation:** Recursive nature can make it complex to implement.

Solutions:

- **In-Place Variants:** Use in-place merge sort variants to reduce space usage.
- **Iterative Version:** Implement an iterative version to avoid deep recursion issues.

Quick Sort

```
package A1;
import java.util.Scanner;
public class QuickSort {
private static int partition(int[] array, int low, int high) {
int pivot = array[high];
int i = low - 1;
for (int j = low; j < high; j++) {
if (array[j] <= pivot) {
i++;
int temp = array[i];
array[i] = array[j];
array[j] = temp;
}
}
int temp = array[i + 1];
array[i + 1] = array[high];
array[high] = temp;
return i + 1;
}
private static void quickSort(int[] array, int low, int high) {
if (low < high) {
int pi = partition(array, low, high);
quickSort(array, low, pi - 1);
quickSort(array, pi + 1, high);
}
}
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.print("Enter the number of elements: ");
int n = scanner.nextInt();
int[] numbers = new int[n];
System.out.println("Enter the elements:");
for (int i = 0; i < n; i++) {
numbers[i] = scanner.nextInt();
}
quickSort(numbers, 0, n - 1);
System.out.println("Sorted array:");
for (int num : numbers) {
System.out.print(num + " ");
}
scanner.close();
}
}
```

Challenges:

- **Worst-Case Time Complexity:** Quick Sort has a worst-case time complexity of O(n^2), which occurs when the pivot is always the smallest or largest element.
- **Pivot Selection:** Choosing a poor pivot can degrade performance, especially with already sorted or nearly sorted arrays.

Solutions:

- **Randomized Pivoting:** Choose a random pivot or use techniques like the Median-of-Three method to improve the likelihood of balanced partitions.
- **Tail Recursion Optimization:** Use tail recursion optimization or iterative approaches to avoid stack overflow in languages that have limited stack space.

Selection sort

```
package A1;
import java.util.Scanner;
public class SelectionSort {
private static void selectionSort(int[] array) {
```

```
int n = array.length;
for (int i = 0; i < n - 1; i++) {
int minIndex = i;
for (int j = i + 1; j < n; j++) {
if (array[j] < array[minIndex]) {
minIndex = j;
}
}
int temp = array[minIndex];
array[minIndex] = array[i];
array[i] = temp;
}
}
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.print("Enter the number of elements: ");
int n = scanner.nextInt();
int[] numbers = new int[n];
System.out.println("Enter the elements:");
for (int i = 0; i < n; i++) {
numbers[i] = scanner.nextInt();
}
selectionSort(numbers);
System.out.println("Sorted array:");
for (int num : numbers) {
System.out.print(num + " ");
}
scanner.close();
}
}
```

Challenges:

- **Time Complexity:** Selection Sort has a time complexity of O(n^2), which makes it inefficient for large datasets.
- **In-Place Sorting:** While Selection Sort is in-place, it still involves multiple swaps and comparisons.

Solutions:

- **Optimized Swaps:** Although Selection Sort cannot avoid O(n^2) time complexity, it can be made slightly more efficient by minimizing swaps.

Shell Sort

```
package A1;
import java.util.Scanner;
public class ShellSort {
```

```
private static void shellSort(int[] array) {
int n = array.length;
for (int gap = n / 2; gap > 0; gap /= 2) {
for (int i = gap; i < n; i++) {
int temp = array[i];
int j = i;
while (j >= gap && array[j - gap] > temp) {
array[j] = array[j - gap];
j -= gap;
}
array[j] = temp;
}
}
}
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.print("Enter the number of elements: ");
int n = scanner.nextInt();
int[] numbers = new int[n];
System.out.println("Enter the elements:");
for (int i = 0; i < n; i++) {
numbers[i] = scanner.nextInt();
}
shellSort(numbers);
System.out.println("Sorted array:");
for (int num : numbers) {
System.out.print(num + " ");
}
scanner.close();
}
}
```

Greedy Algorithm

```
package A1;
import java.util.Arrays;
import java.util.Comparator;
import java.util.Scanner;
class Item {
double value, weight;
Item(double value, double weight) {
this.value = value;
this.weight = weight;
}
}
public class FractionalKnapsack {
public static double getMaxValue(Item[] items, double capacity) {
Arrays.sort(items, new Comparator<Item>() {
@Override
public int compare(Item i1, Item i2) {
return Double.compare((i2.value / i2.weight), (i1.value / i1.weight));
}
});
double totalValue = 0.0;
for (Item item : items) {
if (capacity == 0) break;
double weightToTake = Math.min(item.weight, capacity);
totalValue += weightToTake * (item.value / item.weight);
capacity -= weightToTake;
}
return totalValue;
}
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.print("Enter the number of items: ");
int n = scanner.nextInt();
Item[] items = new Item[n];
for (int i = 0; i < n; i++) {
System.out.print("Enter value and weight for item " + (i + 1) + ": ");
double value = scanner.nextDouble();
double weight = scanner.nextDouble();
items[i] = new Item(value, weight);
}
System.out.print("Enter the capacity of the knapsack: ");
double capacity = scanner.nextDouble();
double maxValue = getMaxValue(items, capacity);
System.out.println("Maximum value in Knapsack = " + maxValue);
scanner.close();
}
}
```

Dijkstra's Algorithm & Bellman-Ford Algorithm

Dijkstra's Algorithm: I use Dijkstra's algorithm to find the shortest path from a source vertex to all other vertices in a graph. I prioritize vertices with the smallest known distance and update the distances of their neighbors. I maintain a priority queue to efficiently fetch the vertex with the minimum distance. I stop when all vertices have been processed or when the shortest path to the destination vertex has been found.

Bellman-Ford Algorithm: I use Bellman-Ford algorithm to find the shortest paths in a graph with possible negative edge weights. I iteratively relax all edges, ensuring that the shortest path to each vertex is updated. I perform this relaxation process for a total of (V-1) times, where V is the number of vertices. I also check for negative weight cycles by performing an additional relaxation and detecting any changes in distances.

Graph Traversals (Breadth-First Search, Depth-First Search)

Breadth-First Search (BFS): I use BFS to explore a graph level by level starting from a given source node. I utilize a queue to keep track of the nodes to visit next. I mark nodes as visited as I encounter them to avoid reprocessing. I can use BFS to find the shortest path in an unweighted graph by counting the number of edges.

Depth-First Search (DFS): I use DFS to explore as far as possible along each branch before backtracking. I utilize a stack or recursion to keep track of the nodes to explore. I mark nodes as visited to prevent infinite loops. I can use DFS to perform tasks such as topological sorting, finding strongly connected components, and solving puzzles like mazes.

Divide and Conquer
I use the divide and conquer strategy to solve problems by breaking them down into smaller subproblems. I divide the problem into smaller, more manageable parts. I solve each subproblem independently. I combine the solutions of the subproblems to obtain the solution to the original problem. I use this approach in algorithms like merge sort, quicksort, and binary search.

Fractional Knapsack Problem
Implementation: I implement the fractional knapsack problem by using a greedy approach. I first calculate the value-to-weight ratio for each item. I then sort the items based on this ratio in descending order. I iteratively add items to the knapsack, starting with the highest ratio, and include fractions of items if necessary to maximize the total value.

Challenges and Solution: I face the challenge of deciding how to handle items that cannot be included in full due to weight constraints. I address this by allowing fractional parts of items to be included, unlike the 0/1 knapsack problem where items are either fully included or excluded. I ensure that the implementation efficiently handles sorting and fractional inclusion to achieve optimal results.